Applicative Matching Logic

Xiaohong Chen
xc3@illinois.edu
University of Illinois at Urbana-Champaign

Grigore Roşu
grosu@illinois.edu
University of Illinois at Urbana-Champaign

July 27, 2019

Abstract

This paper proposes a logic for programming languages, which is both simple and expressive, to serve as a foundation for language semantics frameworks. Matching μ-logic has been recently proposed as a unifying foundation for programming languages, specification and verification. It has been shown to capture several logics important for programming languages, including first-order logic with least fixpoints, separation logic, temporal logics, modal μ-logic, and importantly, reachability logic, a language-independent logic for program verification that subsumes Hoare logic. This paper identifies a fragment of matching μ-logic called *applicative matching logic (AML)*, which is much simpler and thus more appealing from a foundational perspective, yet as expressive as matching μ-logic. Several additional logical frameworks fundamental for programming languages are shown to be faithfully captured by AML, including many- and order-sorted algebras, λ-calculus, (dependent) type systems, evaluation contexts, and rewriting. Finally, it is shown how all these make AML an appropriate underlying logic foundation for complex language semantics frameworks, such as $\mathbb{K}$.

1 Introduction

In an ideal language framework, all programming languages must have formal definitions and all language tools are automatically derived, correct-by-construction, at no additional costs (see Fig. 1). As one of many efforts in pursuing this ideal scenario (see Section 12 for related work), the $\mathbb{K}$ framework (`www.kframework.com`) has been used to define the complete formal semantics of many real-world languages such as C [27], Java [7], JavaScript [44], x86 [17], as well as emerging blockchain languages such as EVM [28] and IELE [30]. All language tools such as parsers, interpreters, and program verifiers are automatically generated from the formal semantics.

In this paper we are interested in logic foundations for language frameworks. For concreteness, we pick $\mathbb{K}$ as a target: in addition to having been used to formalize a variety of real programming languages completely, $\mathbb{K}$ appears to also be one of the most complex tool-supported language framework in use and has no agreed-upon formal semantics. At a high level, every language definition L defines a *logic theory* Γ^L in some foundational logic. Different language tools for L represent different best-effort implementations of logic reasoning within Γ^L. For example, the program verifier can do reasoning of the form $\Gamma^L \vdash \varphi_{\text{init}} \Rightarrow \varphi_{\text{final}}$, which intuitively means that all initial program configurations φ_{init} must reach the final configurations φ_{final}.

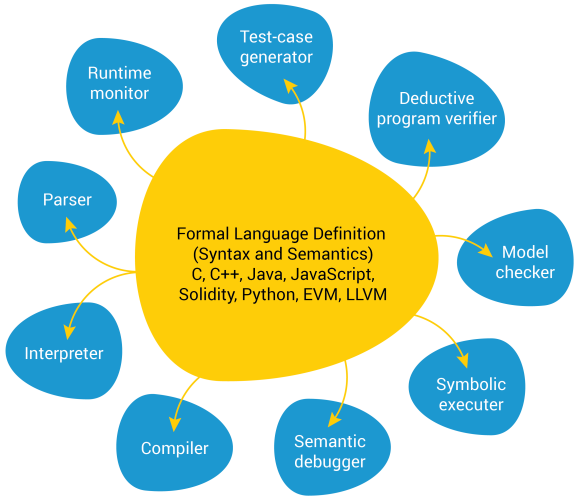

Figure 1: Ideal language framework

1.1 Limitations of matching μ-logic

A major research question is: What logic can serve as such a foundational logic for complex language frame-

works such as $\mathbb{K}$, whose implementation is larger than 130,000 LOC? Previously, $\mathbb{K}$ used a combination of two logics to serve as the foundation for its program verification tools: *matching logic* [48] (shortened as ML) used to specify static structures and logical constraints; and *reachability logic* [50] (shortened as RL) used to reason about dynamic reachability properties. RL has a language-independent proof system that supports sound and relatively complete verification for all programming languages. It is an expressive logic that subsumes Hoare logic and Hoare-style program verification [49], but it cannot express some dynamic properties such as liveness properties. We are aware of three other attempts to give a formal semantics to $\mathbb{K}$: one using (double-pushout) graph transformations [53], another based on a translation to Isabelle [33], and another based on a translation to (coinduction in) Coq [39]. None of these were incorporated within $\mathbb{K}$'s codebase, because none of them are satisfactory: not only they result in heavy translations with a big representational distance from the original definition, but also they support only some properties (e.g., reachability, or partial correctness, or coinductive) or only some of $\mathbb{K}$'s features. In particular, none supports $\mathbb{K}$'s reasoning modulo (evaluation and configuration) contexts in its full generality, and none of them properly captures $\mathbb{K}$'s local rewriting (details in Section 10).

To overcome these limitations, *matching μ-logic* [10] (shortened as MmL) was recently proposed. As a unifying logic, MmL subsumes not only ML and RL, but also many important logical frameworks such as first-order logic (shortened as FOL) and FOL with least fixpoints [25], separation logic [47, 43], modal μ-logic [32], many variants of temporal logic such as linear temporal logic and computation tree logic [46], and dynamic logic [20, 26]. Therefore, MmL is a good logic foundation candidate. However, MmL suffers from at least two main limitations.

Firstly, MmL is more complex than necessary. As a *many-sorted* logic, MmL has theories that can contain multiple, sometimes infinitely many *sorts*, each with its own carrier set in models. Theories define *many-sorted symbols* $\sigma \in \Sigma_{s_1 \dots s_n, s}$ that take a fixed number n of MmL formulas, called *patterns*, of appropriate sorts, and produce patterns of sort s. This places a burden on implementations, which need to store the sorts and the arities of all symbols, carry out well-formedness checking, and implement a more complex than needed proof system and checker.

Secondly, MmL enforces a strict separation among elements, sorts, and symbols. Intuitively, elements represent data; sorts represent the types of data; and symbols represent operations or predicates over data. Therefore, MmL distinguishes among data, types, and operations/predicates. This can become inconvenient when we define, e.g., functional programming languages, where functions are first-class citizens and can be passed around as normal data, or *λ-calculi*, where no distinction is made between data and functions. Here is a concrete example. Suppose we have a sort *Nat* of natural numbers and we want to define *parametric lists*. Specifically, we need to define for every sort s a new sort denoted $\mathit{List}\{s\}$ for all lists over elements of sort s. Then, we can define the common (parametric) operations over (parametric) lists and their axioms in the following usual way (only one axiom is shown; $\Sigma_{\epsilon, \mathit{List}\{s\}}$ denotes the set of all constant symbols of sort $\mathit{List}\{s\}$):

$$\mathit{nil}\{s\} \in \Sigma_{\epsilon, \mathit{List}\{s\}} \qquad \mathit{cons}\{s\} \in \Sigma_{s\, \mathit{List}\{s\}, \mathit{List}\{s\}} \qquad \mathit{append}\{s\} \in \Sigma_{\mathit{List}\{s\}\, \mathit{List}\{s\}, \mathit{List}\{s\}}$$
$$\mathit{append}\{s\}(\mathit{cons}\{s\}(x\colon s, l\colon \mathit{List}\{s\}), l'\colon \mathit{List}\{s\}) = \mathit{cons}\{s\}(x\colon s, \mathit{append}\{s\}(l\colon \mathit{List}\{s\}, l'\colon \mathit{List}\{s\}))$$

Note that all symbols and axioms are parametric in sort s. In other words, there are *infinitely many* sorts: *Nat*, $\mathit{List}\{\mathit{Nat}\}$, $\mathit{List}\{\mathit{List}\{\mathit{Nat}\}\}$, etc., as well as infinitely many symbols and axioms in the MmL theory of parametric lists, even though all of them are highly homogeneous. This is at best inconvenient for MmL implementations. Either we incorporate parametric lists as built-in into the implementations, or we invent some ad-hoc meta-level notation to specify the infinite theory of parametric lists in some finite way. Neither approach is optimal: the former lacks generality while the latter is heavy and superficial. Most many-sorted FOL systems forgo parametricity all together.

1.2 Applicative matching logic: simpler and yet more flexible

Our main contribution is the proposal of *applicative matching logic (AML)* as a logic foundation for programming language semantic frameworks, like $\mathbb{K}$, and thus as a logic foundation for programming languages, program specification and program reasoning in general. We show that AML subsumes MmL and thus it subsumes all the logic frameworks subsumed by MmL. Then we show several new results of how AML subsumes other logics/algebras, including order-sorted algebra, parametric sorts/types, function sorts/types, dependent sorts/types, λ-calculus, and pure type systems; these are defined as theories or notations in AML, which are shown to entail the same theorems as the original logic/calculus/algebra. Finally, we put everything together and show by example how AML, with a proof checker that was implemented in ˜100 statements (in Maude), can serve as a trusted logic foundation for programming

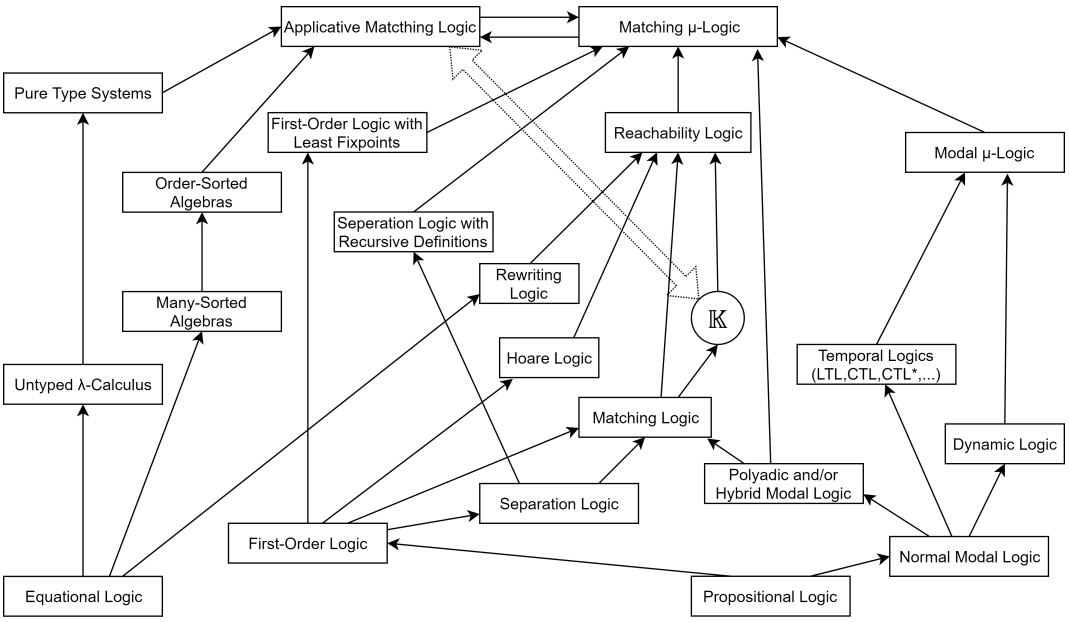

Figure 2: Many important logics, calculi, and algebras can be defined as theories and/or notations in AML; the existing $\mathbb{K}$ implementation (denoted as the node labeled "$\mathbb{K}$") only realizes a fragment of RL and will eventually be lifted to the same level as AML, as denoted by the dotted bidirectional arrow.

language frameworks: we define an AML theory of *contexts* as a generalization of λ-calculus and use it to give a semantics to $\mathbb{K}$. These are illustrated in Fig. 2, where the arrow $\longrightarrow$ is read "can be defined in" or "is subsumed by".

We designed AML by learning from the pros and cons of MmL. Specifically, we drop the many-sorted infrastructure from MmL and keep only the minimal necessary components: a dummy sort denoted as $\star$ for all patterns, a binary symbol $\_\_ \in \Sigma_{\star\star,\star}$ called *application*, and a few constant symbols in $\Sigma_{\epsilon,\star}$. In other words, AML is the fragment of MmL over the above simple signature.

We emphasize that we think of MmL and AML as two different *methodologies*, instead of merely two different *logics*. Indeed, since AML is a fragment of MmL as a logic, then of course all logics subsumed by AML are trivially subsumed by MmL, too. What we find appealing about AML is that it inspires more elegant definitions than MmL; for example, parametric lists as a *finite* AML theory (see Section 7.1), overcoming the inconvenience of the infinite MmL theory (see Section 1.1).

The rest of the paper is organized as follows. We present the syntax, semantics, and proof system of AML in Section 2. Then we define many-sorted algebra, MmL, constructors and term algebras, and order-sorted algebra in AML in Sections 3, 4, 5, and 6, respectively. In Section 7, we show by example how to define parametric sorts/types, function sorts/types, and dependent sorts/types. In Sections 8 and 9 we define λ-calculus and pure type systems, respectively. We propose AML as a logic foundation of $\mathbb{K}$ in Section 10. Finally, we discuss our implementation of an AML proof checker in Section 11, discuss related and future work in Section 12, and conclude with Section 13.

Appendix contains proofs for all the results in the paper.

2 Applicative Matching Logic: Basic Definitions and Notations

Here we introduce the basic definitions and notations of AML.

2.1 Applicative matching logic syntax and semantics

Definition 1. A *signature* is a triple $(\text{EVar}, \text{SVar}, \Sigma)$ with a set EVar of *element variables* $x, y, \ldots$, a set SVar of *set variables* $X, Y, \ldots$, and a set Σ of *constant symbols* or *constants* $\sigma, f, g, \ldots$. We omit EVar and SVar when they are understood and simply use Σ to denote the signature. The set of Σ-*patterns* is inductively defined as follows:

$$\varphi ::= x \in \text{EVar} \mid X \in \text{SVar} \mid \sigma \in \Sigma \mid \varphi_1\ \varphi_2 \mid \bot \mid \varphi_1 \to \varphi_2 \mid \exists x.\varphi \mid \mu X.\varphi \quad \text{if } \varphi \text{ is positive in } X$$

where φ is positive in X if all free occurrences of X in φ are under an even number of negations, where for counting negations we regard $\varphi_1 \to \varphi_2$ as $\neg\varphi_1 \vee \varphi_2$. Let Pattern be the set of all patterns.

The pattern $\varphi_1\ \varphi_2$ is called an *application*. As a convention, application is associative to the left and thus we write $\varphi_1\ \varphi_2\ \varphi_3\ \ldots\ \varphi_n$ instead of $(\cdots((\varphi_1\ \varphi_2)\ \varphi_3)\ \cdots\ \varphi_n)$. The other constructs are standard logical constructs as in FOL and/or modal μ-logic [32]. Element variables are like FOL variables that range over elements in the models, while set variables are like propositional variables in modal logic that range over sets (i.e., predicates); Definition 4 defines the precise semantics of all these constructs. Both $\exists$ and μ are binders and their scope goes as far as possible to the right. The notions of free variables, α-renaming, and capture-avoiding substitution are defined as usual. We write $\text{FV}(\varphi)$ to denote the set of all free (element and set) variables in φ. We write $\varphi[\psi/x]$ (resp. $\varphi[\psi/X]$) to denote the result of substituting all free occurrences of x (resp. X) in φ for ψ, where α-renaming happens implicitly to prevent variable capturing. Note that $\exists$ only binds element variables and μ only binds set variables. We define the following derived constructs:

$$\neg\varphi \equiv \varphi \to \bot \qquad \varphi_1 \vee \varphi_2 \equiv \neg\varphi_1 \to \varphi_2 \qquad \varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2) \qquad \varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1)$$
$$\top \equiv \neg\bot \qquad \forall x.\varphi \equiv \neg\exists x.\neg\varphi \qquad \nu X.\varphi \equiv \neg\mu X.\neg\varphi[\neg X/X] \ \text{// if } \varphi \text{ is positive in } X$$

We assume the standard precedence of these connectives, and application $\varphi_1\varphi_2$ binds the tightest.

In models, patterns evaluate to the sets of elements that *match* them. Intuitively, x is matched by exactly one-element sets (i.e. singletons) while X is matched by any sets; $\bot$ is matched by no elements; $\varphi_1 \to \varphi_2$ is matched by those elements which match φ_2 if they match φ_1 (in particular, all those that do not match φ_1). The pattern $\exists x.\varphi$ allows us to abstract away irrelevant parts (i.e., x) of the pattern φ. The pattern $\varphi_1\varphi_2$ allows us to apply φ_1 that represents a function/operation/predicate, to φ_2 that represents the argument. The pattern $\mu X.\varphi$ evaluates to the smallest set X w.r.t. inclusion such that $X = \varphi$ (note that X may occur free in φ). This intuition is formalized in Definition 4.

Derived constructs have the expected semantics, too. The pattern $\neg\varphi$ is matched by elements not matching φ; $\varphi_1 \vee \varphi_2$ is matched by elements matching φ_1 or φ_2; $\varphi_1 \wedge \varphi_2$ is matched by elements matching both φ_1 and φ_2; $\top$ is matched by all elements; and $\nu X.\varphi$ evaluates to the largest set X such that $X = \varphi$. This is formalized in Proposition 5.

Now, we formally define the semantics of AML. Firstly, we review the following key result.

Theorem 2 (Knaster-Tarski [57])**.** *For any nonempty set M and a monotone function $\mathcal{F} : \mathcal{P}(M) \to \mathcal{P}(M)$, i.e., $\mathcal{F}(A) \subseteq \mathcal{F}(B)$ for all $A \subseteq B$, where $\mathcal{P}(M)$ denotes the powerset of M, we have that $\mathcal{F}$ has a unique least fixpoint $\mu\mathcal{F}$ and a unique greatest fixpoint $\nu\mathcal{F}$, given as:*

$$\mu\mathcal{F} = \bigcup\{A \subseteq M \mid \mathcal{F}(A) \subseteq A\} \qquad\qquad \nu\mathcal{F} = \bigcap\{A \subseteq M \mid A \subseteq \mathcal{F}(A)\}$$

We call A a pre-fixpoint *of $\mathcal{F}$ whenever $\mathcal{F}(A) \subseteq A$ and a* post-fixpoint *of $\mathcal{F}$ whenever $A \subseteq \mathcal{F}(A)$.*

Since patterns evaluate to sets, AML adopts a *powerset semantics*; in particular, application $\varphi_1\varphi_2$ is interpreted as a relation instead of a function.

Definition 3. Given a signature Σ, a Σ-*model* is a triple $(M, \_\bullet\_, \{\sigma_M\}_{\sigma\in\Sigma})$ containing:

- a nonempty carrier set M;
- a binary function $\_\bullet\_ \colon M \times M \to \mathcal{P}(M)$ called *application*;
- an interpretation $\sigma_M \subseteq M$ for every constant $\sigma \in \Sigma$ as a subset of M.

By abuse of notation, we use the same letter M to also denote the model itself.

For notational simplicity, we extend *pointwisely* the application $\_\cdot\_$ over sets as follows:

$$\_\cdot\_ : \mathcal{P}(M) \times \mathcal{P}(M) \to \mathcal{P}(M) \qquad A \cdot B = \textstyle\bigcup_{a\in A, b\in B} a \cdot b \quad \text{for all } A, B \subseteq M.$$

Note $A \cdot B = \emptyset$ if $A = \emptyset$ or $B = \emptyset$. We abbreviate $a \cdot b$ as ab and write $ab_1 \cdots b_n \equiv (\cdots((ab_1)\, b_2)\cdots b_n)$.

AML models generalize *applicative structures* [1], which are pairs $(A, \_\cdot_A\_)$ with a nonempty set A and an *application function* $\_\cdot_A\_ \colon A \times A \to A$. Indeed, applicative structures are special instances of AML models with $|a \cdot b| = 1$ for all $a, b \in M$.

Definition 4. Given a model M, an *M-valuation* is a function $\rho \colon (\text{EVar} \cup \text{SVar}) \to (M \cup \mathcal{P}(M))$ with $\rho(x) \in M$ for all $x \in \text{EVar}$ and $\rho(X) \subseteq M$ for all $X \in \text{SVar}$. That is, element variables evaluate to elements and set variables to sets. Its *extension*, $\bar{\rho} \colon \textsc{Pattern} \to \mathcal{P}(M)$, is defined as:

$$\begin{array}{lll}
\bar{\rho}(x) = \{\rho(x)\} \text{ for all } x \in \text{EVar} & \bar{\rho}(\sigma) = \sigma_M \text{ for all } \sigma \in \Sigma & \bar{\rho}(\bot) = \emptyset \\
\bar{\rho}(X) = \rho(X) \text{ for all } X \in \text{SVar} & \bar{\rho}(\varphi_1\varphi_2) = \bar{\rho}(\varphi_1)\bar{\rho}(\varphi_2) & \bar{\rho}(\varphi_1 \to \varphi_2) = (M \setminus \bar{\rho}(\varphi_1)) \cup \bar{\rho}(\varphi_2) \\
\bar{\rho}(\exists x.\varphi) = \bigcup_{a\in M} \overline{\rho[a/x]}(\varphi) & \bar{\rho}(\mu X.\varphi) = \mu\mathcal{F}^{\rho}_{\varphi,X} \text{ with } \mathcal{F}^{\rho}_{\varphi,X}(A) = \overline{\rho[A/X]}(\varphi) \text{ for } A \subseteq M &
\end{array}$$

where "$\setminus$" denotes set difference; $\rho[a/x]$ (resp. $\rho[A/X]$) denotes the valuation ρ' such that $\rho'(x) = a$ (resp. $\rho'(X) = A$) and agrees with ρ on all other variables; $\mu\mathcal{F}^{\rho}_{\varphi,X}$ denotes the least fixpoint of $\mathcal{F}^{\rho}_{\varphi,X}$.

Note that $\mathcal{F}^{\rho}_{\varphi,X}$ defined as above is a monotone function, so Theorem 2 applies and the least fixpoint indeed exists. Therefore, patterns can evaluate to the empty set $\emptyset$, the total set M, or any other subset $A \subseteq M$. The patterns φ for which $\bar{\rho}(\varphi)$ is either $\emptyset$ or M for every ρ are called *predicates*.

Proposition 5. *Under the above notation, the following hold:*

$$\begin{array}{lll}
\bar{\rho}(\neg\varphi) = M \setminus \bar{\rho}(\varphi) & \bar{\rho}(\varphi_1 \vee \varphi_2) = \bar{\rho}(\varphi_1) \cup \bar{\rho}(\varphi_2) & \bar{\rho}(\varphi_1 \wedge \varphi_2) = \bar{\rho}(\varphi_1) \cap \bar{\rho}(\varphi_2) \\
\bar{\rho}(\top) = M & \bar{\rho}(\varphi_1 \leftrightarrow \varphi_2) = M \setminus (\bar{\rho}(\varphi_1) \triangle \bar{\rho}(\varphi_2)) & \bar{\rho}(\forall x.\varphi) = \bigcap_{a\in M} \overline{\rho[a/x]}(\varphi) \\
\bar{\rho}(\nu X.\varphi) = \nu\mathcal{F}^{\rho}_{\varphi,X} & \textit{with } \mathcal{F}^{\rho}_{\varphi,X} \textit{ defined the same as in Definition 4} &
\end{array}$$

where "$\triangle$" denotes set symmetric difference: $A \triangle B = (A \setminus B) \cup (B \setminus A)$.

Note that the predicate-ness of patterns is preserved by all the core and derived constructs, i.e., $\varphi_1 \to \varphi_2$ and $\varphi_1 \wedge \varphi_2$ are predicates whenever φ_1 and φ_2 are, $\mu X.\varphi$ is a predicate whenever φ is, etc.

Definition 6. We write $M \vDash \varphi$ iff $\bar{\rho}(\varphi) = M$ for all ρ. Let Γ be a set of patterns called *axioms*. We write $M \vDash \Gamma$ iff $M \vDash \psi$ for all $\psi \in \Gamma$, and $\Gamma \vDash \varphi$ iff $M \vDash \varphi$ for all $M \vDash \Gamma$. A *theory* is a pair (Σ, Γ) with a signature Σ and a set of Σ-patterns Γ as axioms. We often omit Σ and use Γ to denote the theory.

2.2 Definedness and related notations

Here we show how to define several important mathematical instruments, such as equality, membership, and functions, as theories in AML. We also introduce notations for them.

Definition 7. Let $\lceil\_\rceil$ be a constant called *definedness*. We write $\lceil\varphi\rceil \equiv \lceil\_\rceil\, \varphi$ and define the axiom:

$$\text{(Definedness)} \qquad \lceil x \rceil$$

We define *totality* $\lfloor\_\rfloor$, *equality* $=$, *membership* $\in$, and *set inclusion* $\subseteq$ as derived constructs as follows:

$$\lfloor\varphi\rfloor \equiv \neg\lceil\neg\varphi\rceil \qquad \varphi_1 = \varphi_2 \equiv \lfloor\varphi_1 \leftrightarrow \varphi_2\rfloor \qquad x \in \varphi \equiv \lceil x \wedge \varphi\rceil \qquad \varphi_1 \subseteq \varphi_2 \equiv \lfloor\varphi_1 \to \varphi_2\rfloor$$

We also define $\varphi_1 \neq \varphi_2 \equiv \neg(\varphi_1 = \varphi_2)$, $x \notin \varphi \equiv \neg(x \in \varphi)$, $\varphi_1 \not\subseteq \varphi_2 \equiv \neg(\varphi_1 \subseteq \varphi_2)$.

We tacitly assume definedness and the above notations in all theories given in this paper, although it is important to note that this is *not* an extension of AML, but simply a constant and an axiom that we assume in subsequent AML theories. Intuitively, the pattern $\lceil\varphi\rceil$ is a predicate that states that φ is *defined*, i.e., φ is matched by some elements. Formally, let M be a model satisfying (Definedness). For notational simplicity, we write $\lceil a\rceil_M \equiv \lceil\_\rceil_M \cdot a$. Since M satisfies (Definedness), $M \vDash \lceil x\rceil$, then $\bar{\rho}(\lceil x\rceil) = \lceil\bar{\rho}(x)\rceil_M = \lceil\rho(x)\rceil_M = M$ for all valuations ρ, which implies that

$\lceil a \rceil_M = M$ for all $a \in M$. Therefore, if $\bar{\rho}(\varphi) \neq \emptyset$, there exists some $a \in \bar{\rho}(\varphi)$ and then $\bar{\rho}(\lceil \varphi \rceil) = \lceil \bar{\rho}(\varphi) \rceil_M \supseteq \lceil a \rceil_M = M$, which implies that $\bar{\rho}(\lceil \varphi \rceil) = M$. Otherwise, if $\bar{\rho}(\varphi) = \emptyset$, then $\bar{\rho}(\lceil \varphi \rceil) = \lceil \bar{\rho}(\varphi) \rceil_M = \lceil \emptyset \rceil_M = \emptyset$.

The following proposition shows that the derived constructs in Definition 7 are all predicates and have the expected semantics: $\lfloor \varphi \rfloor$ checks if φ is *total*, i.e., if φ is matched by all elements; $\varphi_1 = \varphi_2$ checks if φ_1 and φ_2 are matched by the same elements; $x \in \varphi$ checks if the element matching x also matches φ; and $\varphi_1 \subseteq \varphi_2$ checks if all elements matching φ_1 also match φ_2.

Proposition 8. *With the above notation, the following hold:*

- $\bar{\rho}(\lceil \varphi \rceil) = M$ *iff* $\bar{\rho}(\varphi) \neq \emptyset$*; and* $\bar{\rho}(\lceil \varphi \rceil) = \emptyset$ *iff* $\bar{\rho}(\varphi) = \emptyset$*;*
- $\bar{\rho}(\lfloor \varphi \rfloor) = M$ *iff* $\bar{\rho}(\varphi) = M$*; and* $\bar{\rho}(\lfloor \varphi \rfloor) = \emptyset$ *iff* $\bar{\rho}(\varphi) \neq \emptyset$*;*
- $\bar{\rho}(\varphi_1 = \varphi_2) = M$ *iff* $\bar{\rho}(\varphi_1) = \bar{\rho}(\varphi_2)$*; and* $\bar{\rho}(\varphi_1 = \varphi_2) = \emptyset$ *iff* $\bar{\rho}(\varphi_1) \neq \bar{\rho}(\varphi_2)$*;*
- $\bar{\rho}(x \in \varphi) = M$ *iff* $\rho(x) \in \bar{\rho}(\varphi)$*; and* $\bar{\rho}(x \in \varphi) = \emptyset$ *iff* $\rho(x) \notin \bar{\rho}(\varphi)$*;*
- $\bar{\rho}(\varphi_1 \subseteq \varphi_2) = M$ *iff* $\bar{\rho}(\varphi_1) \subseteq \bar{\rho}(\varphi_2)$*; and* $\bar{\rho}(\varphi_1 \subseteq \varphi_2) = \emptyset$ *iff* $\bar{\rho}(\varphi_1) \not\subseteq \bar{\rho}(\varphi_2)$*.*

As seen in Definitions 3 and 4, AML has a powerset semantics where patterns evaluate to sets. In particular, application $\varphi_1 \varphi_2$ and constant symbols are interpreted as relations. In the following, we show that it is easy to "recover" the classic functional semantics of application and constants.

Proposition 9. *Let σ be any constant symbol. We define the following two axioms:*

$$\text{(Functional Constant)}\ \ \exists z . \sigma = z \qquad \text{(Functional Application)}\ \ \exists z . xy = z$$

Then, for any model M satisfying (Functional Constant)*, σ_M is a singleton set, and for any model M satisfying* (Functional Application)*, its application is a function, i.e., $|a \cdot b| = 1$ for all $a, b \in M$.*

To avoid writing such functional axioms repeatedly, we simply say σ is a *functional constant* to mean that we automatically assume the axiom (Functional Constant) for σ.

As an example, we show that *applicative structures* [1, Definition 5.1.1] and their special instances, *combinatory algebras* [52, 16], can be axiomatically defined in AML. Combinatory algebras are of particular importance in the study of the foundation of mathematics and computation, because they yield an equivalent formalization of *λ-calculus* [51]. We will study λ-calculus and its AML definition in Section 8. Here, we only discuss applicative structures and combinatory algebras, as they are simple, yet interesting AML examples.

Definition 10 ([1, Definition 5.1.1, Definition 5.1.7])**.** An *applicative structure* is a pair $(A, \_\cdot_A\_)$ containing a nonempty set A and a binary function $\_\cdot_A\_ \colon A \times A \to A$ called application. The structure A is a *combinatory algebra* if there are two distinguished elements $k, s \in A$ such that $k \cdot_A a \cdot_A b = a$ and $s \cdot_A a \cdot_A b \cdot_A c = a \cdot_A b \cdot_A (a \cdot_A c)$, for all $a, b, c \in A$.

Proposition 11. *Any AML model satisfying* (Functional Application) *is an applicative structure. Additionally, if the AML theory includes two functional constants $k, s \in \Sigma$ satisfying the axioms $kxy = x$ and $sxyz = xy(xz)$, then its models are combinatory algebras.*

2.3 Applicative matching logic proof system

Here we present the proof system of AML. We first define *application contexts*.

Definition 12. An *application context* C is a pattern with a distinguished placeholder variable $\Box$ such that the path from the root of C to $\Box$ has only applications. We write "$C[\varphi] \equiv C[\varphi/\Box]$".

We show a Hilbert-style proof system of AML in Fig. 3. The proof system is obtained by instantiating the proof system of MmL given in [10, Fig. 1] over the signature of AML, i.e., the signature containing one sort, one binary symbol, and some constants. We denote the corresponding provability relation as $\Gamma \vdash \varphi$, which means that φ can be proved by the proof system with patterns in Γ taken as additional axioms. We abbreviate $\emptyset \vdash \varphi$ as $\vdash \varphi$.

(Propositional Tautology)	φ if φ is a propositional tautology over patterns
(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
($\exists$-Quantifier)	$\varphi[y/x] \to \exists x.\varphi$
($\exists$-Generalization)	$\dfrac{\varphi_1 \to \varphi_2}{(\exists x.\varphi_1) \to \varphi_2}$ if $x \notin \mathrm{FV}(\varphi_2)$
(Propagation$_\bot$)	$C[\bot] \to \bot$
(Propagation$_\vee$)	$C[\varphi_1 \vee \varphi_2] \to C[\varphi_1] \vee C[\varphi_2]$
(Propagation$_\exists$)	$C[\exists x.\varphi] \to \exists x.C[\varphi]$ if $x \notin \mathrm{FV}(C)$
(Framing)	$\dfrac{\varphi_1 \to \varphi_2}{C[\varphi_1] \to C[\varphi_2]}$
(Set Variable Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$
(Pre-Fixpoint)	$\varphi[\mu X.\,\varphi/X] \to \mu X.\,\varphi$
(Knaster-Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{\mu X.\,\varphi \to \psi}$
(Existence)	$\exists x.x$
(Singleton)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$

Figure 3: The Hilbert-style proof system of AML (where C, C_1 and C_2 are application contexts)

The proof rules in Fig. 3 can be divided into four categories. The first category contains the first four rules that provide *complete FOL reasoning* [55]. The second category contains four rules that provide *frame reasoning* over application contexts. The third category contains three rules that provide standard *fixpoint reasoning* as in modal μ-logic [32]. Finally, the last two technical rules are needed for certain completeness results [10].

The following proposition shows that *equational reasoning* is also sound in AML for theories that contain definedness, with which equality is defined as a derived construct (see Definition 7).

Proposition 13. *For all theories* Γ *with definedness, the following hold:*

$\Gamma \vdash \varphi = \varphi$ $\qquad$ $\Gamma \vdash \varphi_1 = \varphi_2$ *and* $\Gamma \vdash \varphi_2 = \varphi_3$ *implies* $\Gamma \vdash \varphi_1 = \varphi_3$

$\Gamma \vdash \varphi_1 = \varphi_2$ *implies* $\Gamma \vdash \varphi_2 = \varphi_2$ $\qquad$ $\Gamma \vdash \varphi_1 = \varphi_2$ *implies* $\Gamma \vdash \psi[\varphi_1/x] = \psi[\varphi_2/x]$.

In conclusion, FOL reasoning, frame reasoning, fixpoint reasoning, and equational reasoning (for theories with definedness) are all available in AML. The following *soundness theorem* shows that all the above reasoning implies semantic validity in the models.

Theorem 14 (Soundness Theorem)**.** $\Gamma \vdash \varphi$ *implies* $\Gamma \models \varphi$.

3 Instance: Many-sorted algebra

Here we show how to define many-sorted algebra (shortened as MSA) in AML. At a high level, we define for every sort s in MSA a corresponding constant in AML, also denoted s, which represents the *sort name*. Then we define a special constant $[\![\_]\!]$, called *inhabitant set*, which can be applied to s and the result $[\![\_]\!]\ s$, written $[\![s]\!]$, is a pattern matched by precisely all elements of sort s. In other words, $[\![s]\!]$ denotes the inhabitant set of s. We will use the same method to define order-sorted algebra in Section 6 and other more complex sort/type structures in Section 7.

Definition 15 ([38, Definition 1])**.** A *many-sorted signature* is a pair (S, Σ) with a nonempty sort set S and an $(S^* \times S)$-indexed set $\Sigma = \{\Sigma_{s_1 \ldots s_n, s}\}_{s_1,\ldots,s_n,s \in S}$ of *many-sorted functions*. We write $f\colon s_1 \times \cdots \times s_n \to s$ to mean that $f \in \Sigma_{s_1 \ldots s_n, s}$.

An (S, Σ)-MSA is a pair $M = (\{M_s\}_{s \in S}, \{f_M\}_{f \in \Sigma})$, consisting of a nonemptyset carrier set M_s for each $s \in S$ and a (total function) interpretation $f_A \colon M_{s_1} \times \cdots \times M_{s_n} \to M_s$ for each $f \in \Sigma_{s_1 \ldots s_n, s}$.

Now we define an AML theory Γ^{MSA} that faithfully captures (S, Σ)-MSA. We follow the classic *sort-as-predicate* paradigm (see, e.g., [14, Section 5]), but thanks to the powerset semantics of AML (Definition 4), our definition Γ^{MSA} is more succinct.

For each sort $s \in S$, we define a corresponding functional constant s representing its *name*. By abuse of language, the constant s is also called a *sort* in AML. The key ingredient is the *inhabitant set* constant $[\![\_]\!]$. The pattern $[\![s]\!]$ denotes the inhabitant set of s. For example, $[\![Nat]\!]$ is all natural numbers, $[\![List]\!]$ is all finite lists, and $[\![Cfg]\!]$ is all program configurations. Strictly speaking, $[\![\_]\!]$ can be applied to any patterns, including meaningless ones that do not represent sorts.

Properties about sorts can be specified as patterns. For example, $x \in [\![s]\!]$ checks if x is in the inhabitant set of s, i.e., if x has sort s. Similarly, $\varphi \subseteq [\![s]\!]$ checks if all elements matching φ have sort s. In MSA, all sorts have nonempty carrier sets. This can be captured by the following axiom:

$$\text{(Nonempty Sort)} \quad [\![s]\!] \neq \bot$$

For each many-sorted function $f \in \Sigma_{s_1 \ldots s_n, s}$, we define a corresponding constant f and use the application pattern $f x_1 \ldots x_n$ to capture the term $f(x_1, \ldots, x_n)$ in MSA. We define the following axiom to specify that f is indeed a function from $s_1, \ldots, s_n$ to s:

$$\text{(Function)} \quad x_1 \in [\![s_1]\!] \land \cdots \land x_n \in [\![s_n]\!] \to \exists y.\, y \in [\![s]\!] \land f x_1 \cdots x_n = y$$

We introduce the *functional notation* $f \colon s_1 \times \cdots \times s_n \to s$ to mean that we automatically assume the axiom (Function) for f. Finally, we let Γ^{MSA} be the theory that contains all the above axioms.

In the following, we connect the AML models of Γ^{MSA} with MSA. Note that AML models make no distinction among elements, sorts, and functions, so they contain not only elements of MSA, but also their sorts and the many-sorted functions. To compare with MSA, we need to first *restrict* AML models over the many-sorted signature (S, Σ). This is formalized below.

Definition 16. Let $(M, \_\bullet\_, \{\sigma_M\}_{\sigma \in \Sigma^{\mathsf{MSA}}})$ be an AML model with $M \vDash \Gamma^{\mathsf{MSA}}$. Its *restricted model w.r.t.* (S, Σ) is an MSA $M^r = (\{M^r_s\}_{s \in S}, \{f_{M^r}\}_{f \in \Sigma})$ that consists of:

- a carrier set $M^r_s = [\![s_M]\!]_M$ for every $s \in S$, where we write $[\![A]\!]_M \equiv [\![\_]\!]_M \bullet A$ for all $A \subseteq M$;
- an interpretation $f_{M^r} \colon M^r_{s_1} \times \cdots \times M^r_{s_n} \to M^r_s$ for every $f \in \Sigma_{s_1 \ldots s_n, s}$, defined such that $\{f_{M^r}(a_1, \ldots, a_n)\} = f_M a_1 \cdots a_n$ for all $a_1 \in M^r_{s_1}, \ldots, a_n \in M^r_{s_n}$.

Note that M^r as given above is a well-defined (S, Σ)-MSA. Its carrier set M^r_s is nonempty because of the (Nonempty Sort) axiom. The interpretation f_{M^r} is a function because of the (Function) axiom, which forces f_M to return singletons on all arguments of appropriate sorts.

Theorem 17. *(S, Σ)-MSA are exactly the restricted Γ^{MSA}-models w.r.t. (S, Σ).*

Sometimes, it is convenient to define a new constant *Sort* to represent the sort set, containing all sorts. This is especially true when we encounter more complex sort/type structures, like in Sections 6 and 7. In MSA, sorts are isolated, so we only need the following axiom for each $s \in S$:

$$\text{(Sort)} \quad s \in \mathit{Sort}$$

to specify that s is a sort. If S is finite, we may instead use only one axiom $\mathit{Sort} = \bigvee_{s \in S} s$ to define precisely *Sort*, but it is not possible when S is infinite. Axiom (Sort) yields more *modular* and *extensible* theories. In practice, a theory is often an aggregation of many sub-theories (e.g., the theory of program configurations contains the theories of numbers, lists, heaps, etc.). Then, every sub-theory can define its own sorts separately, making the entire theory more modular. It is also more extensible. For example, adding a new sort $s^* \notin S$ only needs to add one more axiom $s^* \in \mathit{Sort}$, without modifying other axioms. On the other hand, the axiom $\mathit{Sort} = \bigvee_{s \in S} s$ summarizes all sorts at once, making it easy to look up a sort, but it is less modular, and adding new sorts requires changing the axiom itself. Both styles have their advantages and we will use both in this paper.

3.1 Example: natural numbers (without induction)

As an example, we define an AML theory Γ^{NAT0} that captures natural numbers. Here we do not consider the inductive principle of natural numbers, which will be handled in Section 5.

Let *Sort* be a constant, representing the sort set. Let *Nat* be a sort, defined by axiom $Nat \in Sort$. We define the following self-explanatory many-sorted functions about natural numbers:

$$0 : \to Nat \qquad succ : Nat \to Nat \qquad plus : Nat \times Nat \to Nat \qquad mult : Nat \times Nat \to Nat$$

These functions can be axiomatized in the usual way. For example, the following axiom specifies that any natural number plus zero equals itself:

$$\forall x . x \in [\![Nat]\!] \to plus\ x\ 0 = x.$$

Note the condition $x \in [\![Nat]\!]$ cannot be omitted. Recall that in AML, models make no distinction between elements, sorts, and functions. Without $x \in [\![Nat]\!]$, the (wrong) axiom $plus\ x\ 0 = x$ specifies more cases than needed, including those when x is not a natural number. For example, it implies $plus\ plus\ 0 = plus$, which is meaningless. In practice, we often need such *sorted quantification*, so we introduce them as derived constructs, where the notation is borrowed from many-sorted FOL:

$$\exists x : s . \varphi \equiv \exists x . x \in [\![s]\!] \wedge \varphi \qquad\qquad \forall x : s . \varphi \equiv \forall x . x \in [\![s]\!] \to \varphi$$

The following proposition shows that sorted $\exists/\forall$-quantification are dual to each other, as expected.

Proposition 18. *Under the above notations,* $\vdash \forall x : s . \varphi = \neg \exists x : s . \neg \varphi$ *and* $\vdash \exists x : s . \varphi = \neg \forall x : s . \neg \varphi$.

Using sorted quantification, we can rewrite the above axiom about *plus* more compactly as:

$$\forall x : Nat . plus\ x\ 0 = x$$

We omit the other usual axioms that define *zero*, *succ*, *plus*, and *mult* and denote the resulting theory as Γ^{NAT0}. Of course, Γ^{NAT0} is by no means a complete axiomatization. In particular, it does not specify the inductive principle of natural numbers. We will complete it in Section 5.

4 Instance: Matching μ-Logic

Here we show how to define matching μ-logic (MmL) [10] in AML, exiling some details to Appendix D. MmL is a many-sorted FOL variant that makes no distinction between function and predicate symbols, uniformly using them to build patterns extended with direct support for least fixpoints. MmL patterns are indexed by sorts, and MmL models are a variant of many-sorted FOL models where all symbols are interpreted as relations of appropriate sort arity.

Definition 19 ([10, Definitions 17,19])**.** A *matching μ-logic signature* is a triple (S, V, Σ) with a many-sorted signature (S, Σ) and $V = EV \cup SV$, a disjoint union of two sets of *sorted variables*, where $EV = \{EV_s\}_{s \in S}$ contains *sorted element variables* $x : s, y : s, \ldots$ and $SV = \{SV_s\}_{s \in S}$ contains *sorted set variables* $X : s, Y : s, \ldots$. (S, V, Σ)*-patterns* are defined for all $s, s' \in S$ as follows:

$$\varphi_s ::= x : s \mid X : s \mid \sigma(\varphi_{s_1}, \ldots, \varphi_{s_n}) \text{ where } \sigma \in \Sigma_{s_1 \ldots s_n, s} \mid \varphi_s \wedge \varphi'_s \mid \neg \varphi_s \mid \exists x : s' . \varphi_s \mid \mu X : s . \varphi_s$$

where $\mu X : s . \varphi_s$ requires all free occurrences of $X : s$ are under an even number of negations in φ_s. The notions of substitution, α-renaming, etc., are defined as usual. An (S, V, Σ)*-model* is a pair $M = (\{M_s\}_{s \in S}, \{\sigma_M\}_{\sigma \in \Sigma})$, consisting of a nonempty carrier set M_s for every $s \in S$ and an interpretation $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_s)$ for every $\sigma \in \Sigma_{s_1 \ldots s_n, s}$. We extend σ_M to its *pointwise extension*, $\sigma_M : \mathcal{P}(M_{s_1}) \times \cdots \times \mathcal{P}(M_{s_n}) \to \mathcal{P}(M_s)$, defined as $\sigma_M(A_1, \ldots, A_n) = \bigcup_{a_i \in A_i, 1 \le i \le n} \sigma(a_1, \ldots, a_n)$ for $A_i \subseteq M_{s_i}$, $1 \le i \le n$. An M-valuation $\rho : V \to M \cup \mathcal{P}(M)$ is one such that $\rho(x : s) \in M_s$ and $\rho(X : s) \subseteq M_s$ for all $x : s, X : s \in V$. Its *extension* $\bar{\rho}$ interprets (S, V, Σ)-patterns to sets as follows:

$$\begin{array}{ll} \bar{\rho}(x : s) = \{\rho(x : s)\} & \bar{\rho}(\sigma(\varphi_{s_1}, \ldots, \varphi_{s_n})) = \sigma_M(\bar{\rho}(\varphi_{s_1}), \ldots, \bar{\rho}(\varphi_{s_n})) \text{ for all } \sigma \in \Sigma_{s_1 \ldots s_n, s} \\ \bar{\rho}(X : s) = \rho(X : s) & \bar{\rho}(\varphi_s \wedge \varphi'_s) = \bar{\rho}(\varphi_s) \cap \bar{\rho}(\varphi'_s) \quad \bar{\rho}(\exists x : s' . \varphi_s) = \bigcup_{a \in M_{s'}} \overline{\rho[a/x : s']}(\varphi_s) \\ \bar{\rho}(\neg \varphi_s) = M \setminus \bar{\rho}(\varphi_s) & \bar{\rho}(\mu X : s . \varphi_s) = \mu \mathcal{F}^{\rho}_{\varphi, X : s} \text{ with } \mathcal{F}^{\rho}_{\varphi, X : s}(A) = \overline{\rho[A/X : s]}(\varphi) \text{ for } A \subseteq M_s \end{array}$$

We write $M \vDash_{\mathsf{MmL}} \varphi_s$ iff $\bar{\rho}(\varphi) = M_s$ for all ρ, and if Ω is an (S, V, Σ)-theory then we write $\Omega \vDash_{\mathsf{MmL}} \varphi_s$ iff $M \vDash_{\mathsf{MmL}} \varphi_s$ for all MmL models with $M \vDash_{\mathsf{MmL}} \Omega$.

Now we define the AML theory Γ^{MmL} that captures the (S, V, Σ)-models of MmL. As in Section 3, we define for each sort $s \in S$ a corresponding functional constant and use the axiom $[\![s]\!] \neq \bot$ to specify that its inhabitant set is nonempty. For every $\sigma \in \Sigma_{s_1 \dots s_n, s}$, we define a corresponding constant and use the following axiom to specify its arity:

$$(\textsc{Arity}) \quad \sigma\, [\![s_1]\!]\, \dots\, [\![s_n]\!] \subseteq [\![s]\!]$$

Intuitively, (Arity) says that if σ is applied to arguments of appropriate sorts, then the result, which can be any set instead of a singleton, is included in the inhabitant set of s. Note that (Arity) is weaker than (Function) in Section 3, as the latter requires additionally that σ produces singletons.

Let Γ^{MmL} contain all the above definitions and axioms.

Definition 20. Let $(M, \_\cdot\_, \{\sigma_M\}_{\sigma \in \Sigma^{\mathsf{MSA}}})$ be an AML model with $M \vDash \Gamma^{\mathsf{AML}}$. Its *restricted model w.r.t.* (S, V, Σ)[1] is an (S, V, Σ)-model of MmL, written $M^r = (\{M^r_s\}_{s \in S}, \{\sigma_{M^r}\}_{s \in S})$, that consists of:

- a carrier set $M^r_s = [\![s_M]\!]_M$ for $s \in S$;
- an interpretation $\sigma_{M^r}(a_1, \dots, a_n) = \sigma_M a_1 \cdots a_n$ for $\sigma \in \Sigma_{s_1 \dots s_n, s}$, $a_1 \in M^r_{s_1}, \dots, a_n \in M^r_{s_n}$.

Theorem 21. *(S, V, Σ)-models of MmL are exactly the restricted Γ^{MmL}-models w.r.t. (S, V, Σ).*

In the following, we show how to translate any MmL pattern φ to a semantically equivalent AML pattern, written φ^{AML}. We assume w.l.o.g. that for each $x{:}s \in EV$ (resp. $X{:}s \in SV$), there is a corresponding $x^s \in \textsc{EVar}$ (resp. $X^s \in \textsc{SVar}$), where s is simply a decoration. We define φ^{AML} as:

$$(x{:}s)^{\mathsf{AML}} \equiv x^s \quad (X{:}s)^{\mathsf{AML}} \equiv X^s \quad (\varphi_s \wedge \varphi'_s)^{\mathsf{AML}} \equiv \varphi^{\mathsf{AML}}_s \wedge \varphi'^{\mathsf{AML}}_s \quad (\neg\varphi_s)^{\mathsf{AML}} \equiv \neg_s \varphi^{\mathsf{AML}}_s$$
$$(\sigma(\varphi_{s_1}, \dots, \varphi_{s_n}))^{\mathsf{AML}} \equiv \sigma \varphi^{\mathsf{AML}}_{s_1} \cdots \varphi^{\mathsf{AML}}_{s_n} \quad \exists x{:}s\,.\,\varphi_s \equiv \exists x^s{:}s\,.\,\varphi^{\mathsf{AML}}_s \quad \mu X{:}s\,.\,\varphi_s \equiv \mu X^s\,.\,\varphi^{\mathsf{AML}}_s$$

where $\neg_s \varphi^{\mathsf{AML}}_s \equiv \neg\varphi^{\mathsf{AML}}_s \wedge [\![s]\!]$, called *sorted negation*, guarantees that MmL pattern sorts are preserved when translated to AML. Finally, we define $\varphi^{\mathsf{VALID}}_s \equiv \psi \rightarrow (\varphi^{\mathsf{AML}}_s = [\![s]\!])$ to specify that φ_s is valid (as an MmL pattern), where $\psi \equiv \bigwedge_{x^{s'} \in \mathrm{FV}(\varphi^{\mathsf{AML}}_s)} x^{s'} \in [\![s']\!] \wedge \bigwedge_{X^{s'} \in \mathrm{FV}(\varphi^{\mathsf{AML}}_s)} X^{s'} \subseteq [\![s']\!]$ specifies that all variables are of the decorated sorts. If Ω is an (S, V, Σ)-theory, let Ω^{VALID} be $\Gamma^{\mathsf{MmL}} \cup \{\varphi^{\mathsf{VALID}} \mid \varphi \in \Omega\}$.

Theorem 22. *Under the notations of Definition 20 and the above MmL-to-AML translation, any M^r-valuation ρ of MmL derives an M-valuation ρ^{AML} of AML, with $\rho^{\mathsf{AML}}(x^s) = \rho(x{:}s)$ and $\rho^{\mathsf{AML}}(X^s) = \rho(X{:}s)$. Furthermore, $\overline{\rho^{\mathsf{AML}}}(\varphi^{\mathsf{AML}}_s) = \bar{\rho}(\varphi_s)$ for all ρ; and $\Omega^{\mathsf{VALID}} \vDash \varphi^{\mathsf{VALID}}_s$ iff $\Omega \vDash_{\mathsf{MmL}} \varphi_s$ for all Ω.*

5 Instance: Constructors and term algebras

Constructors are extensively used in building programs and data, as well as semantic structures to define and reason about languages and programs. They generate *term algebras*, whose elements are terms and functions are constructors that build terms. In the broader context of defining formal semantics of languages, term algebras, as a special case of *initial algebras*, play an important role and have led to many applications and tools (see [23] for more on initial algebra semantics; for OSA-based tools, see OBJ [24] and Maude [13]). In this section, we show how to define constructors and term algebras in AML; in particular, we use the least fixpoint μ-binder to define axioms that support *inductive reasoning* in term algebras.

Let us fix a many-sorted signature $(\{Term\}, C)$ with one sort *Term* and a set C of functions, called *constructors*, where at least one of them is a constant. The same technique applies to multiple sorts. We use $c, d, \dots$ to denote constructors. The set of *C-terms*, denoted as T^C_{Term}, is defined as follows:

$$\text{terms } t ::= c \in C_{\epsilon, Term} \text{ “constant terms”} \quad | \quad c(t_1, \dots, t_n) \text{ for } c \in C_{Term\,\dots\,Term, Term} \text{ “compound terms”}$$

A *C-term algebra* is a $(\{Term\}, C)$-MSA $T^C = (T^C_{Term}, \{c_{T^C}\}_{c \in C})$, with T^C_{Term} as carrier set and interpretations $c_{T^C} : T^C_{Term} \times \cdots \times T^C_{Term} \rightarrow T^C_{Term}$ for $c \in C_{Term \dots Term, Term}$, s.t. $c_{T^C}(t_1, \dots, t_n) = c(t_1, \dots, t_n)$.

[1] We use (S, V, Σ) to distinguish from the restricted model for MSA in Definition 16; technically speaking, V is not needed.

Now we define an AML theory Γ^{TERM} that faithfully captures the term algebra T^C. Since T^C is a MSA, we let Γ^{TERM} contain all axioms for MSA as in Section 3. Then, we specify that (1) all constructors are injective functions; and (2) the inhabitant set of *Term* is precisely T^C_{Term}, as below:

(No Confusion I, for all dinstinct $c, d \in C$)

$\forall x_1 : Term \cdots \forall x_n : Term . \forall y_1 : Term \cdots \forall y_n : Term . c x_1 \cdots x_n \neq d y_1 \cdots y_m$

(No Confusion II, for all $c \in C$)

$\forall x_1 : Term \cdots \forall x_n : Term . \forall x'_1 : Term \cdots \forall x'_n : Term . c x_1 \cdots x_n = c x'_1 \cdots x'_n \rightarrow x_1 = x'_1 \wedge \cdots \wedge x_n = x'_n$

(Inductive Domain) $[\![Term]\!] = \mu D . \bigvee_{c \in C} c D \cdots D$ // as many D's as the arity of c

Intuitively, (No Confusion I) says different constructors build different terms; (No Confusion II) says that constructors are injective functions; (Inductive Domain) forces that $[\![Term]\!]$ is the smallest set closed under all constructors, yielding exactly T^C_{Term}, as shown in the following proposition.

Proposition 23. *If $M \models \Gamma^{\mathsf{TERM}}$ then the restricted model M^r w.r.t. $(\{Term\}, C)$ is isomorphic to T^C.*

The proof follows from the proof of [10, Proposition 22], which there was done for the more general context of matching μ-logic. Note that AML can define term algebras *up-to-isomorphism*, and not only up-to-elementary-equivalence as with FOL [34].

5.1 Example: Natural numbers (with induction)

We continue the definition of natural numbers in Section 3.1, by extending Γ^{NAT0} with the above constructor axioms for *zero* and *succ*. As an example, (Inductive Domain) takes the form:

$$\text{(Inductive Domain)} \quad [\![Nat]\!] = \mu D . zero \vee succ\ D$$

We denote the resulting theory as Γ^{NAT}.

Now we show that the Peano induction axiom of the (first-order) Peano arithmetic [37, 45] can be proved as a *theorem* in Γ^{NAT}, using the proof system of AML. Recall that Peano arithmetic is a FOL theory of natural numbers with addition and multiplication, where formulas are built from equalities and FOL connectives. Since both equalities and FOL connectives are subsumed in AML notationally, Peano arithmetic formulas are well-formed AML patterns. Here we show the Peano induction axiom, where $\varphi(x)$ is a FOL formula with a distinguished variable x:

$$\text{(Peano Induction)} \quad \varphi(0) \wedge (\forall y . \varphi(y) \rightarrow \varphi(succ(y))) \rightarrow \forall x . \varphi(x)$$

Note that the above is an axiom schema, defined for each $\varphi(x)$, due to the limitation of FOL, which has no variables that range over sets/predicates. In AML, however, we can use set variables to range over all sets (without quantification) and replace the schema with *one* axiom:

Proposition 24. $\Gamma^{\mathsf{NAT}} \vdash zero \in X \wedge (\forall y : Nat . y \in X \rightarrow succ\ y \in X) \rightarrow \forall x : Nat . x \in X$.

Intuitively, a set X that contains *zero* and is closed under *succ* contains all natural numbers. Note that all instances of (Peano Induction) can be proved as theorems from this proposition, by letting $\Psi \equiv \exists z : Nat . z \wedge \varphi(z)$ be the pattern matched by exactly all natural numbers z such that $\varphi(z)$ holds. By standard AML reasoning, we can prove that $(x \in \Psi) = \varphi(x)$. Then we can reduce (Peano Induction) to the proposition by applying (Set Variable Substitution) in Fig. 3.

6 Instance: Order-sorted algebra

Here we show how to define *order-sorted algebra* (shortened as OSA) in AML. OSA extends MSA with a partial ordering on sorts, called *subsort relation* forces a corresponding *subset relation* on the carrier sets. In addition, OSA allows *overloaded functions* with different arities.[2]

[2]There are many OSA variants. Here we consider *overloaded-OSA*, as defined in [38], for concreteness, but the same techniques can be applied to other variants.

Definition 25 ([38, Definition 1])**.** An *order-sorted signature* is a triple $(S, \leq, \Sigma)$ with a nonempty sort set S, a subsort relation $\leq \subseteq S \times S$, and an $(S^* \times S)$-indexed set $\Sigma = \{\Sigma_{s_1 \dots s_n, s}\}_{s_1, \dots, s_n, s \in S}$ of *order-sorted functions*, whose elements are denoted $f^{s_1 \dots s_n, s} \in \Sigma_{s_1 \dots s_n, s}$. We assume the *monotonicity condition*: if $f^{s_1 \dots s_n, s} \in \Sigma_{s_1 \dots s_n, s}$ and $f^{s'_1 \dots s'_n, s'} \in \Sigma_{s'_1 \dots s'_n, s'}$ with $s_1 \leq s'_1, \dots, s_n \leq s'_n$ called *subsort overloading*, then $s \leq s'$. For simplicity, we only consider subsort overloading. An $(S, \leq, \Sigma)$-OSA is a pair $M = (\{M_s\}_{s \in S}, \{f_M^{s_1 \dots s_n, s}\}_{f^{s_1 \dots s_n, s} \in \Sigma})$ that consists of:

- a nonempty carrier set M_s for each sort $s \in S$ with $M_s \subseteq M_{s'}$ for all $s \leq s'$;
- an interpretation $f_M^{s_1 \dots s_n, s} \colon M_{s_1} \times \cdots \times M_{s_n} \to M_s$ for each $f \in \Sigma_{s_1 \dots s_n, s}$ with $f_M^{s_1 \dots s_n, s} = f_M^{s'_1 \dots s'_n, s'} \restriction_{M_{s_1} \times \cdots \times M_{s_n}}$ for all $s_1 \leq s'_1, \dots, s_n \leq s'_n$, where "$\restriction$" denotes function restriction.

Now we define an AML theory Γ^{OSA} that faithfully captures OSA. Compared with MSA, we just need to handle the subsort relation and the overloaded functions. For the former, we define:

$$\text{(Subsort)} \quad [\![s]\!] \subseteq [\![s']\!] \quad \text{for all sorts } s, s' \in S \text{ with } s \leq s'.$$

which specifies that the inhabitant set of s is a subset of the inhabitant set of s'. For the overloaded functions, we define a set of constants $\{f \mid f^{s_1 \dots s_n, s} \in \Sigma\}$ and use one AML constant f to represent all its overloaded copies in OSA. Then, we can specify that f is indeed a function of the appropriate arity as we do for MSA, but now every f can have multiple axioms, one for each of its overloaded copies. Specifically speaking, we define the following axiom for every $f^{s_1 \dots s_n, s} \in \Sigma_{s_1 \dots s_n, s}$ (note that we use the sorted quantification notation defined in Section 3.1):

$$\text{(Function)} \quad \forall x_1 : s_1 \dots \forall x_n : s_n \,.\, \exists y : s \,.\, f x_1 \cdots x_n = y$$

By abuse of notation, we also write $f \colon s_1 \times \cdots \times s_n \to s$ to mean the above (Function) axiom for f.

Let Γ^{OSA} be the theory containing all the above axioms. Now we connect AML models of Γ^{OSA} with OSA. As in MSA, we need to first restrict the AML models, as formalized below:

Definition 26. Let $(M, \_\bullet\_, \{\sigma_M\}_{\sigma \in \Sigma^{\mathsf{OSA}}})$ be an AML model with $M \models \Gamma^{\mathsf{OSA}}$. Its *restricted model w.r.t.* $(S, \leq, \Sigma)$ is an OSA $M^r = (\{M^r_s\}_{s \in S}, \{f_{M^r}^{s_1 \dots s_n, s}\}_{f^{s_1 \dots s_n, s} \in \Sigma})$ that consists of:

- a carrier set $M^r_s = [\![s_M]\!]_M$ for every $s \in S$;
- an interpretation $f_{M^r}^{s_1 \dots s_n, s} \colon M^r_{s_1} \times \cdots \times M^r_{s_n} \to M^r_s$, defined such that $\{f_{M^r}^{s_1 \dots s_n, s}(a_1, \dots, a_n)\} = f_M a_1 \cdots a_n$ for all $a_1 \in M^r_{s_1}, \dots, a_n \in M^r_{s_n}$.

Note that M^r as given above is a well-defined $(S, \leq, \Sigma)$-OSA. Its subset relation is enforced by (Subsort). The subsort-overloaded functions $f_{M^r}^{s_1 \dots s_n, s}$ and $f_{M^r}^{s'_1 \dots s'_n, s'}$ coincide on their overlapped part, because they are both defined by the same interpretation f_M.

Theorem 27. *$(S, \leq, \Sigma)$-OSA are exactly the restricted Γ^{OSA}-models w.r.t. $(S, \leq, \Sigma)$.*

It is known (see, e.g., [22]) that OSA can be defined in (many-sorted) FOL, where the subsort relation is captured by the *coercion functions* $c^{s'}_s \in \Sigma_{s,s'}$ for all $s \leq s'$. Intuitively, $c^{s'}_s$ denotes the embedding from sort s to sort s'. This approach, however, is not practically useful, as noticed in [40, pp. 9]. We explain why by an example. Suppose there are three sorts $s \leq s' \leq s''$, a constant a of sort s, and a function $f \in \Sigma_{s'',s''}$. Then, the term $f(x)$ has multiple parses when translated to FOL, e.g.: $f(c^{s''}_s(a))$ and $f(c^{s''}_{s'}(c^{s'}_s(a)))$. This means that all tools for OSA based on FOL need to do reasoning *modulo* the triangle property $c^{s''}_s(a) = c^{s''}_{s'}(c^{s'}_s(a))$, which is inconvenient and causes huge overhead. In contrast, Γ^{OSA} is more succinct. Both the subsort relation and subsort overloading can be axiomatized, and no coercion functions are needed.

7 Examples of More Complex Sort/Type Structures

Here we show, by examples, how to use the same method we used for MSA/OSA to define more complex sort/type structures, such as parametric sorts, function sorts, and dependent sorts/types.

7.1 Example of parametric sorts/types: Parametric lists

Let us extend Γ^{NAT}, the theory of natural numbers given in Section 5.1, to Γ^{LIST}, the theory of *parametric lists*. A parametric list, denoted *List s*, is the sort of finite lists with elements of sort *s*. For example, *List Nat* is the lists of natural numbers, *List* (*List Nat*) is the lists of lists of naturual numbers, etc. Therefore, we define a function $List\colon Sort \to Sort$, called *sort constructor*, which takes a sort *s* and produces the sort *List s* of lists parametric in *s*. Standard list operations can be defined as functions (recall the notations given in Section 3.1, where $\forall x\colon s.\varphi \equiv \forall x.\, x \in [\![s]\!] \to \varphi$, etc.):

$$\forall s\colon Sort.\, \exists l'\colon List\ s.\, nil = l' \qquad \forall s\colon Sort.\, \forall x\colon s.\, \forall l\colon List\ s.\, \exists l'\colon List\ s.\, cons\ x\ l = l'$$
$$\forall s\colon Sort.\, \forall l_1\colon List\ s.\, \forall l_2\colon List\ s.\, \exists l'\colon List\ s.\, append\ l_1\ l_2 = l'$$

Note that the above axioms are similar to the (Function) axioms that define many-sorted functions (see Section 3), but here *s* is a variable ranging over *Sort*, instead of a sort constant. For simplicity, we take the freedom to use our function notation also under quantifiers, writing the above as:

$$\forall s\colon Sort.\, nil\colon \to List\ s \qquad \forall s\colon Sort.\, cons\colon s \times List\ s \to List\ s \qquad \forall s\colon Sort.\, append\colon List\ s \times List\ s \to List\ s$$

As in term algebras, we can specify the set of lists as the smallest set closed under *nil* and *cons*:

$$\text{(Inductive List)} \quad \forall s\colon Sort.\, [\![List\ s]\!] = \mu L.\, nil \vee cons\ [\![s]\!]\ L$$

Note that $[\![s]\!]$ denotes all elements of sort *s*. (Inductive List) supports *inductive reasoning* about lists. For example, let us define a new sort *Int* and add axioms to specify that *Nat* is a *subsort* of *Int*:

$$\text{(Sort)} \quad Int \in Sort \qquad\qquad \text{(Subsort)} \quad [\![Nat]\!] \subseteq [\![Int]\!].$$

Then we can prove the following; the proof is insightful, e.g., see the use of (Knaster-Tarski):

Proposition 28. *Under the above notations and axioms, we have* $\vdash [\![List\ Nat]\!] \subseteq [\![List\ Int]\!]$.

Proof. By (Inductive List), it suffices to prove $\vdash (\mu L.\, nil \vee cons\ [\![Nat]\!]\ L) \subseteq [\![List\ Int]\!]$. By [11], we can replace "$\subseteq$" with "$\to$": $\vdash (\mu L.\, nil \vee cons\ [\![Nat]\!]\ L) \to [\![List\ Int]\!]$. By (Knaster Tarski), it suffices to prove $\vdash nil \vee cons\ [\![Nat]\!]\ [\![List\ Int]\!] \to [\![List\ Int]\!]$. Since $\vdash [\![Nat]\!] \subseteq [\![Int]\!]$, by frame reasoning, we have $\vdash cons\ [\![Nat]\!]\ [\![List\ Int]\!] \to cons\ [\![Int]\!]\ [\![List\ Int]\!]$. Then by FOL reasoning, it suffices to prove $\vdash nil \vee cons\ [\![Int]\!]\ [\![List\ Int]\!] \to [\![List\ Int]\!]$, which is proved by (Pre-Fixpoint). □

We let Γ^{LIST} be the theory containing all the above axioms.

Note that the set of all parametric sorts also forms a term algebra, generated from the primitive sorts *Nat* and *Int*, as well as the sort constructor *List*. This is specified as follows:

$$\text{(Sorts)} \quad [\![Sort]\!] = \mu S\ .Nat \vee Int \vee List\ S$$

Suppose that we want only lists of the primitive sorts *Nat* and *Int*, forbidding nested lists. This can be achieved by replacing the above (Sorts) axiom with the following:

$$\text{(No Nested Lists)} \quad [\![Sort]\!] = Nat \vee Int \vee List\ Nat \vee List\ Int$$

A more modular way is to introduce a new constant *PrimitiveSort* that contains all primitive sorts and let *Sort* contain only *PrimitiveSort* and *List PrimitiveSort*:

$$PrimitiveSort = Nat \vee Int \qquad\qquad Sort = PrimitiveSort \vee List\ PrimitiveSort.$$

Note that *nil* is overloaded; that is, all lists have the same empty list. If we want *nil* to be also parametric, we can replace the function definition $nil\colon \to List\ s$ with the following axioms:

$$\text{(Parametric Nil)} \quad \forall s\colon Sort.\, nil\ s \in [\![List\ s]\!] \qquad \text{(Functional Nil)} \quad \forall s\colon Sort\ \exists x\colon List\ s.\, nil\ s = x$$

The (Inductive List) axiom is modified accordingly as follows:

$$\text{(Inductive List II)} \quad \forall s\colon Sort.\, [\![List\ s]\!] = \mu L.\, nil\ s \vee cons\ [\![s]\!]\ L \quad /\!/ \text{ when } nil \text{ is parametric in } s$$

To sum up, there are no hard rules and AML gives us a lot of power and flexibility.

7.2 Example of function sorts/types

Here we show how to define *function sorts/types*. Let $\mathit{Function}\colon \mathit{Sort} \times \mathit{Sort} \to \mathit{Sort}$ be a sort constructor (like *List* above); $\mathit{Function}\ s\ s'$ is the sort of all functions from s to s', as specified below:

$$\text{(Function Sort)} \quad \forall s\colon \mathit{Sort}\,.\,\forall s'\colon \mathit{Sort}\,.\,[\![\mathit{Function}\ s\ s']\!] = \exists f\,.\,f \wedge \forall x\colon s\,.\,\exists y\colon s'\,.\,f\,x = y$$

Recall that $\exists$ means set union (see Definition 4). Therefore, $[\![\mathit{Function}\ s\ s']\!]$ is matched by all f's such that $\forall x\colon s\,\exists y\colon s'\,.\,f\,x = y$, i.e., f is a function from s to s'. For notational simplicity, we define

$$\mathit{Function}\ s_1\ \cdots\ s_n\ s \equiv \mathit{Function}\ s_1\ (\mathit{Function}\ s_2\ \cdots\ (\mathit{Function}\ s_n\ s)\cdots)$$

As an example, we extend Γ^{LIST} with two higher-order list operations: *fold* and *map*, which are common in functional programming languages and can be straightforwardly defined as:

$$\begin{array}{ll}
\forall s\colon \mathit{Sort}\,.\,\forall s'\colon \mathit{Sort}\,. & \forall s\colon \mathit{Sort}\,.\,\forall s'\colon \mathit{Sort}\,. \\
\quad \mathit{fold}\colon \mathit{Function}\ s'\ s\ s' \times s' \times \mathit{List}\ s \to s' & \quad \mathit{map}\colon \mathit{Function}\ s\ s' \times \mathit{List}\ s \to \mathit{List}\ s' \\
\forall f\colon \mathit{Function}\ s'\ s\ s'\,.\,\forall x\colon s'\,.\,\mathit{fold}\ f\ x\ \mathit{nil} = x & \forall g\colon \mathit{Function}\ s\ s'\,.\,\mathit{map}\ g\ \mathit{nil} = \mathit{nil} \\
\forall f\colon \mathit{Function}\ s'\ s\ s'\,.\,\forall x\colon s'\,.\,\forall y\colon s\,.\,\forall l\colon \mathit{List}\ s\,. & \forall g\colon \mathit{Function}\ s\ s'\,.\,\forall y\colon s\,.\,\forall l\colon \mathit{List}\ s\,. \\
\quad \mathit{fold}\ f\ x\ (\mathit{cons}\ y\ l) = \mathit{fold}\ f\ (f\,x\,y)\ l & \quad \mathit{map}\ g\ (\mathit{cons}\ y\ l) = \mathit{cons}\ (g\ y)\ (\mathit{map}\ g\ l)
\end{array}$$

7.3 Example of dependent sorts/types: machine integers

We next define *dependent sorts/types*, which are like parametric sorts but they are parametric in elements, instead of other sorts. This makes no big difference in AML, for it makes no distinction between elements, sorts, and operations, all of which are defined uniformly by patterns. Therefore, the same method we use to define parametric sorts can also be applied to define dependent sorts.

As an example, suppose we want to define a dependent sort *MInt*, called *machine integers*, such that $\mathit{MInt}\ n$ for $n \geq 1$ is the sort of machine integers of *size* n, i.e., natural numbers less than 2^n. For clarity, we define a new sort *Size* for positive natural numbers and axiomatize *MInt* as follows:

$$[\![\mathit{Size}]\!] = \mathit{succ}\ [\![\mathit{Nat}]\!] \qquad \mathit{MInt}\colon \mathit{Size} \to \mathit{Sort} \qquad \forall n\colon \mathit{Size}\,.\,[\![\mathit{MInt}\ n]\!] = \exists x\colon \mathit{Nat}\,.\,x \wedge x < \mathit{pow}_2\ n$$

where $\mathit{pow}_2\colon \mathit{Nat} \to \mathit{Nat}$ (power of 2) and $\_<\_$ can be defined in the usual way.

We can define functions over machine integers, such as *mplus* and *mmult*, by defining their arities and then *reusing* the addition *plus* and the multiplication *mult* over natural numbers:

$$\begin{array}{l}
\forall n\colon \mathit{Size}\,.\,\mathit{mplus}\colon \mathit{MInt}\ n \times \mathit{MInt}\ n \to \mathit{MInt}\ (\mathit{succ}\ n) \\
\forall n\colon \mathit{Size}\,.\,\forall x\colon \mathit{MInt}\ n\,\forall y\colon \mathit{MInt}\ n\,.\,\mathit{mplus}\ x\ y = \mathit{plus}\ x\ y \\
\forall n\colon \mathit{Size}\,.\,\forall m\colon \mathit{Size}\,.\,\mathit{mmult}\colon \mathit{MInt}\ n \times \mathit{MInt}\ m \to \mathit{MInt}\ (\mathit{plus}\ n\ m) \\
\forall n\colon \mathit{Size}\,.\,\forall m\colon \mathit{Size}\,.\,\forall m\colon \mathit{Size}\,\forall x\colon \mathit{MInt}\ n\,\forall y\colon \mathit{MInt}\ m\,.\,\mathit{mmult}\ x\ y = \mathit{mult}\ x\ y
\end{array}$$

8 Instance: λ-calculus

Here we show how to faithfully define *untyped λ-calculus* [12] as a theory in AML. As a foundational theory about computation, λ-calculus differs from other formalizations in that it regards functions as *processes*, going from arguments to results [1]. It is also a *higher-order calculus*, where functions are first-class citizens that can be passed as arguments as normal data. There are many variants and extensions of λ-calculus. Here we discuss the one where all λ-terms are of the same sort/type, but will consider a typed extension of λ-calculus in Section 9.

Assume a set Var of *variables* $x, y, \ldots$. The set of *λ-terms*, denoted Λ, is inductively defined as

$$\lambda\text{-terms} \qquad e ::= x \in \textsc{Var} \mid e_1\ e_2 \ \text{ “λ-application”} \mid \lambda x\,.\,e \ \text{ “λ-abstraction”}$$

Here, λ is a binder, just like $\exists$ in AML. The notions of free variables, capture-avoiding substitution, and α-equivalence are defined as usual. In λ-calculus, we are interested in proving equations $e_1 = e_2$, for $e_1, e_2 \in \Lambda$, using the proof

$$\begin{array}{ccccc} \Gamma^\lambda \vdash e_1 = e_2 & \Longrightarrow_1 & \Gamma^\lambda \vDash e_1 = e_2 & \Longrightarrow_2 & M \vDash e_1 = e_2 \text{ for all AML models } M \vDash \Gamma^\lambda \\ & & & & \Downarrow_3 \\ \vdash_\lambda e_1 = e_2 & \Longleftarrow_5 & \vDash_\lambda e_1 = e_2 & \Longleftarrow_4 & M \vDash_\lambda e_1 = e_2 \text{ for all concrete ccc models } M \end{array}$$

Figure 4: A model-theoretic approach to proving completeness of Γ^λ

system of λ-calculus, which includes standard rules for equational reasoning and the following axiom (schema) that specifies the result of function application:

$$(\beta) \quad (\lambda x.e)\, e' = e[e'/x] \quad \text{for all } x \in \textsc{Var} \text{ and } e, e' \in \Lambda$$

We write $\vdash_\lambda e_1 = e_2$ to mean that $e_1 = e_2$ can be proved in λ-calculus.

Our goal is to define an AML theory Γ^λ that faithfully captures λ-calculus. The theory Γ^λ should subsume all λ-calculus syntax, especially the λ-binder, so that all λ-terms and their equations are well-formed patterns in Γ^λ. In addition, we prove the following *conservative extension* result:

$$\Gamma^\lambda \vdash e_1 = e_2 \quad \underset{\text{soundness}}{\overset{\text{completeness}}{\rightleftharpoons}} \quad \vdash_\lambda e_1 = e_2 \qquad \text{for all } e_1, e_2 \in \Lambda$$

This result says that we can safely reduce λ-calculus reasoning to AML reasoning, without worrying that we prove fewer or more equations after the reduction than before it. Specifically, the *soundness* means that all provable equations in λ-calculus can also be proved in Γ^λ, while the *completeness* says that no more equations can be proved. Note that both directions consider only the equations between λ-terms, i.e., patterns of the form $e_1 = e_2$, instead of arbitrary AML patterns.

The rest of the section is organized as follows. We discuss the main challenges in defining Γ^λ in Section 8.1, review the semantics of λ-calculus in Section 8.2, give preparing definitions in Section 8.3, and finally define Γ^λ in Section 8.4. The conservative extension is proved as Theorem 35.

8.1 Main challenges of defining λ-calculus as a theory

There are two main challenges. Firstly, we need to define λ's binding behavior. The key observation is that λ plays two important roles at the same time: (1) it builds a *term* and (2) it builds a *binding* of its first argument into its second. Fortunately, AML allows us to separate these two roles, where we define terms as in MSA/OSA and we define the binding using AML's built-in $\exists$-binder.

The second challenge is to prove the conservative extension. The soundness proof is easy. Since λ-calculus reasoning is just normal equational reasoning plus (β), we just need to let Γ^λ contain all instances of (β). The completeness proof is more involved. Indeed, AML has a richer syntax and a more complex proof system than λ-calculus, but we need to prove that this extra infrastructure cannot be used to prove more equations in AML than in λ-calculus.

We prove the completeness following a *model-theoretic* approach, as illustrated in Fig. 4. Here, we consider a special class of *λ-calculus models*, called *concrete ccc models* [1, Definition 5.5.9], which are given as the reflexive objects of a concrete cartesian closed category. Specifically, we use two important facts about concrete ccc models. Firstly, they are special instances of the AML models of Γ^λ, which is the major requirement to satisfy when defining Γ^λ. Secondly, they are *complete* for λ-calculus, i.e., every valid equation is provable, where an equation is valid iff it holds in all concrete ccc models.[3] These facts are used in Steps 3 and 5 in Fig. 4, respectively.

It is known that λ-calculus is equivalent to combinatory algebras [1, Theorem 7.3.10]. In particular, $\lambda x.e$ can be simulated by k and s, the two distinguished constants in combinatory algebra (see Definition 10). Since combinatory algebra can be defined in AML, we could simply define λ-calculus by firstly translating it to combinatory algebra and then define the latter. We did not choose this approach. Instead, we define λ-calculus *directly*, without any translation. This is because we want to understand how to deal with *binding* in general. Many programming languages have language constructs that create bindings, and it is not practical to develop a translation for each of these languages' binders to some other languages without binder, not to mention proving the translation correct. Therefore, it is necessary for language frameworks (such as $\mathbb{K}$) to have a general and uniform method for dealing with binding constructs.

[3] In literature about λ-calculus, *completeness* of a class of models often means *representability completeness* [5, Definition 2(iii)], i.e., there exists a model whose valid equations are exactly those provable in λ-calculus. Here we use *completeness* in a different, more FOL way, where we consider equations holding in not one model, but all models.

Different models of λ-calculus

We give a brief summary of models of λ-calculus and discuss why we choose to use the concrete ccc models for proving the completeness of our axiomatization Γ^{λ}.

There are mainly three types of models (called *semantic notions*; see [35] for a survey). Firstly, there are λ-*models* [1, Section 5.2], which are combinatory algebras that provide coherent interpretations of all λ-terms. Secondly, there are *categorical models* [1, Section 5.5], which are given as the reflexive objects of cartesian closed categories (shortened as ccc), where λ-terms are interpreted as morphisms. Thirdly, there are *Hindley-Longo models* [29], which form an alternative presentation of λ-models and interpret λ-terms directly, without translating them to combinatory terms. The concrete ccc models used in this paper are categorical models with *strictly concrete categories* [1, Definition 5.5.8].

We choose concrete ccc models because they have a non-categorical set-theoretic presentation [4] that fits well with the powerset semantics of AML. In concrete ccc models, the interpretation of a λ-term is inductively defined from the interpretation of its sub-terms, so it is more natural to turn concrete ccc models into AML models, which is important in proving the completeness, as discussed in Section 8.1. In contrast, both λ-models and Hindley-Longo models interpret λ-terms *all at the same time*, with some side conditions satisfied. For example, in Hindley-Longo models, the interpretation of $\lambda x.e$ under valuation ρ, written $|\lambda x.e|_\rho$, is some *unspecified element* such that $|\lambda x.e|_\rho \cdot a = |e|_{\rho[a/x]}$ for all elements a, while in concrete ccc models, it is explicitly defined as $|\lambda x.e|_\rho = \mathbb{G}(f^\rho_{e,x})$, where $f^\rho_{e,x}(a) = |e|_{\rho[a/x]}$ for all a and $\mathbb{G}$ is the *retraction function* that encodes functions into elements. Therefore, it is more natural to turn concrete ccc models into AML models, because we can have an explicit, constructive interpretation of $\lambda x.e$.

8.2 Concrete ccc models of λ-calculus

Definition 29 ([4, Definition 57])**.** Given an applicative structure $(M, \_\cdot\_)$, its set of *representable functions* is given as $R(M) = \{f\colon M \to M \mid f(x) = a \cdot x \text{ for an } a \in M\}$. A *pre-model* (of λ-calculus) is a triple $(M, \_\cdot\_, \mathbb{G})$, where the *retraction function* $\mathbb{G}\colon R(M) \to M$ satisfies $\mathbb{A} \circ \mathbb{G} = id_{R(M)}$ with $\mathbb{A}\colon M \to R(M)$ defined as $\mathbb{A}(a)(x) = a \cdot x$. Given $\rho\colon \textsc{Var} \to M$, we define $|e|_\rho$ as follows:

$$|x|_\rho = \rho(x) \qquad |e_1 e_2|_\rho = |e_1|_\rho |e_2|_\rho \qquad |\lambda x.e|_\rho = \mathbb{G}(f^\rho_{e,x}) \text{ where } f^\rho_{e,x}(a) = |e|_{\rho[a/x]} \text{ for } a \in M$$

M is a *concrete ccc model* iff $f^\rho_{e,x} \in R(M)$ for all e and ρ, which implies that M interprets all λ-terms. We write $M \vDash_\lambda e_1 = e_2$ iff $|e_1|_\rho = |e_2|_\rho$ for all ρ, and $\vDash_\lambda e_1 = e_2$ iff $M \vDash_\lambda e_1 = e_2$ for all M.

The following lemma is the key result for proving our completeness (i.e., Step 5 in Fig. 4).

Lemma 30 ([31])**.** $\vDash_\lambda e_1 = e_2$ *implies* $\vdash_\lambda e_1 = e_2$.

8.3 Pairs, sets and powersets

As discussed in Section 8.1, one major challenge in defining λ-calculus is to deal with λ-bindings. Recall the interpretation of $\lambda x.e$ in concrete ccc models (Definition 29), namely $|\lambda x.e|_\rho = \mathbb{G}(f^\rho_{e,x})$. To define it, we need to define (1) the retraction function $\mathbb{G}$; and (2) the function $f^\rho_{e,x}$. Notice that $\mathbb{G}$ is one fixed function given directly by the model, while $f^\rho_{e,x}$ is defined constructively from the interpretation of e. We can define $\mathbb{G}$ by a constant G, as we define MSA/OSA functions, but we cannot do that for $f^\rho_{e,x}$. We need to construct its definition from e, and that is the real challenge.

Our solution is to define $f^\rho_{e,x}$ by specifying its *graph*, which is *a set of pairs* defined as:

$$\mathsf{graph}(f^\rho_{e,x}) = \{(a, f^\rho_{e,x}(a)) \mid a \in M\} = \{(a, |e|_{\rho[a/x]}) \mid a \in M\}$$

Therefore, we need to define both *pairs* and *sets*, which we do below.

Note that the notions defined in this section are useful to have in general, not only for λ-calculus.

8.3.1 Defining pairs

Here we define, axiomatically, pairing sorts and constructs:

Definition 31. For any sort s, the *product sort* of s is a new sort denoted as s^2. We define *paring* as a function $\langle \_, \_ \rangle \colon s \times s \to s^2$, write $\langle \_, \_ \rangle\ \varphi_1\ \varphi_2$ as $\langle \varphi_1, \varphi_2 \rangle$, and define the following axioms:

$$\text{(Product Set)} \quad [\![s^2]\!] = \exists x \colon s \,.\, \exists y \colon s \,.\, \langle x, y \rangle$$
$$\text{(Injectivity)} \quad \forall x_1 \colon s \,.\, \forall y_1 \colon s \,.\, \forall x_2 \colon s \,.\, \forall y_2 \colon s \,.\, \langle x_1, y_1 \rangle = \langle x_2, y_2 \rangle \to x_1 = x_2 \wedge y_1 = y_2$$

Intuitively, $\langle x, y \rangle$ denotes the pair (x, y). The (Product Set) axiom specifies that the inhabitant set of s^2 is precisely the product of the inhabitant sets of s, as shown in the following proposition.

Proposition 32. *For all models M satisfying the axioms in Definition 31, $[\![s^2]\!]_M \cong [\![s]\!]_M^2$.*

8.3.2 Defining sets and powersets

AML has a builtin powerset semantics, where a pattern φ is matched by a set X of elements. Here we show how to turn φ into another pattern, denoted *int* φ, which is matched by exactly one element in the powerset: the set X. This helps us to turn graphs such as $\mathsf{graph}(f^{\rho}_{e,x})$ into elements, so that we can apply the retraction $\mathbb{G}$ on them. Using the language of set theory, we call *int* φ the *intension* of φ, but firstly, we define its reverse *ext* φ, called *extension*.

Definition 33. For any sort s, the *power sort* of s is a new sort, written 2^s, whose element variables are written $\alpha, \beta, \ldots$. We define a constant *ext*, called *extension*, and define the following axioms:

$$\text{(Arity)} \quad \forall \alpha \colon 2^s \,.\, ext\ \alpha \subseteq [\![s]\!] \qquad \text{(Powerset)} \quad X \subseteq [\![s]\!] \to \exists \alpha \colon 2^s \,.\, ext\ \alpha = X$$
$$\text{(Extensionality)} \quad \forall \alpha \colon 2^s \,.\, \forall \beta \colon 2^s \,.\, ext\ \alpha = ext\ \beta \to \alpha = \beta$$

Intuitively, the inhabitant set of 2^s is the *powerset* of the inhabitant set of s. In other words, *elements* of sort 2^s are *sets* of elements of sort s, so α and β are effectively ranging over sets (of elements of sort s). The pattern *ext* α has sort s and is matched by all elements in *the unique set* that matches α (recall that α is an element variable of sort 2^s); so semantically, *ext* is interpreted as the identity function over the powerset of the inhabitant set of s, but syntactically, we need it to connect s and 2^s. The axiom (Arity) specifies the arity of *ext*. The axiom (Powerset) says that for all sets X of elements of sort s, there is some element α such that the extension of α is X. Therefore, the inhabitant set of 2^s is *at least* as large as the powerset of the inhabitant set of s. The last axiom (Extensionality) says that two sets α and β are equal if their extensions are equal, making the inhabitant set of 2^s *exactly the same* as the powerset of the inhabitant set of s, as shown below.

Proposition 34. *For all models M satisfying the axioms in Definition 33, $[\![2^s]\!]_M \cong \mathcal{P}([\![s]\!]_M)$.*

Note that we did not use the least fixpoint μ-binder in the above axioms, and yet we capture precisely powersets. Also, we *did* use a set variable X in (Power Set) to range over all possible sets of elements of sort s. Note that we can not write $\forall X$, because $\forall$ binds only element variables, but we can let X be free, and free variables in axioms are implicitly universally quantified, by Definition 4.

Finally, we define the inverse of extension, called *intension*, as the following syntactic sugar:

$$int\ \varphi \equiv \exists \alpha \,.\, \alpha \wedge ext\ \alpha = \varphi$$

Suppose φ is a pattern of sort s. The pattern *int* φ, which has sort 2^s, is matched by the *unique* α such that *ext* $\alpha = \varphi$, where the uniqueness is guaranteed by (Extensionality).

8.4 Defining λ-calculus in applicative matching logic

Now we are ready to define the AML theory Γ^{λ} that faithfully captures λ-calculus. Firstly, we define three sorts: *Term* as the sort of λ-terms; $Pair \equiv Term^2$ as the product sort of *Term*; and $Graph \equiv 2^{Pair}$ as the power sort of *Pair*. Next, we define $G \colon Graph \rightharpoonup Term$ to be a *partial function* as follows:

$$\text{(Partial Function)} \quad \forall g \colon Graph \,.\, G\ g = \bot \vee \exists t \colon Term \,.\, G\ g = t$$

Next, we take all λ-calculus variables as element variables of Γ^{λ} and define λ-application as the built-in AML application. Next, we define λ-abstraction as syntactic sugar:

$$\lambda x \,.\, e \equiv G\ (int\ \exists x \colon Term \,.\, \langle x, e \rangle)$$

Intuitively, $\exists x{:}\mathit{Term}.\langle x, e\rangle$ is the union of all pairs $\langle x, e\rangle$ for all valuations of x; in other words, it is the graph of the function given by e, i.e., $f^{\rho}_{e,x}$ (Definition 29). Then, *int* takes the graph as a set and returns itself as an element in the powerset, which is then passed to G that defines the retraction function $\mathbb{G}$. Finally we let Γ^{λ} contain all the above axioms/notations, plus all instances of (β).

Theorem 35. $\vdash_{\lambda} e_1 = e_2$ *iff* $\Gamma^{\lambda} \vdash e_1 = e_2$.

We discuss the proof intuitively. The soundness (i.e. the "only if" direction) holds because Γ^{λ} contains all instances of (β) and equational reasoning is available in AML (Proposition 13). The completeness (i.e. the "if" direction) is proved following Fig. 4, where the only nontrivial reasoning step is Step 3, which boils down to showing that every concrete ccc model $M = (M, \_\bullet\_, \mathbb{G})$ derives an AML model, written M^{AML}, such that $M^{\mathsf{AML}} \vDash \Gamma^{\lambda}$ and $M^{\mathsf{AML}} \vDash e_1 = e_2$ implies $M \vDash_{\lambda} e_1 = e_2$. Intuitively, M^{AML} is given by taking M as the inhabitant set of *Term*, $M \times M$ as the one of *Pair*, and $\mathcal{P}(M \times M)$ as the one of *Graph*, which includes the graphs of all representable functions in $R(M)$. We take $\mathbb{G}$ to be the interpretation of G, i.e., let $G_{M^{\mathsf{AML}}} \cdot \mathsf{graph}(f) = \mathbb{G}(f)$, for all $f \in R(M)$. Then we can prove $M^{\mathsf{AML}} \vDash \Gamma^{\lambda}$ except (β). Next, we show for any valuation ρ of λ-calculus, $\{|e|_{\rho}\} = \bar{\rho}(e)$ for all $e \in \Lambda$, by structural induction on e (note that ρ is also an AML valuation). Then we know both M^{AML} satisfies (β) and that $M^{\mathsf{AML}} \vDash e_1 = e_2$ implies $M \vDash_{\lambda} e_1 = e_2$, which finishes the proof.

8.5 A generic method for dealing with binders

Note that Theorem 35 holds even if we remove all axioms about pairs and powersets from Γ^{λ}, keeping only (β). Indeed, the completeness holds with fewer axioms, and the soundness holds as long as (β) is included. This important observation suggests that the following syntactic sugar

$$\lambda x.e \equiv G\ (\mathit{int}\ \exists x.\langle x, e\rangle)$$

actually gives us a generic method for dealing with binders. The fact that $\langle \_, \_\rangle$ is paring, *int* is intension, and G is the retraction function, is *irrelevant*, as their axioms are not needed in Theorem 35 at all. What matters is that $\langle \_, \_\rangle$ and G are two constants, and *int* is the reverse of a constant (i.e., *ext*). This inspires us to use the following generic method for defining *any binders*. Suppose we want to define a *binder*, say $\lambda(x, e_1, \ldots, e_n)$, that takes $n+1$ arguments and binds x into $e_1, \ldots, e_n$, we define it as the following syntactic sugar:

$$\lambda(x, e_1, \ldots, e_n) \equiv G\ (\mathit{int}\ \exists x.\langle x, e_1, \ldots, e_n\rangle)$$

where $\langle \ldots\rangle$ and G are constants and *int* is the reverse of a constant *ext* and we write $\langle \ldots\rangle\ \varphi_1 \cdots \varphi_n$ as $\langle \varphi_1, \ldots, \varphi_n\rangle$ for all $n \geq 0$. When there are multiple binders, we use different sets of $\langle \ldots\rangle$, G and *int* (and thus also *ext*) to avoiding confusion. Sometimes, we want to define $\lambda(x, e_1, \ldots, e_n)$ that only binds x into some of $e_1, \ldots, e_n$, not all of them; one typical example is $\texttt{let}\ x = e\ \texttt{in}\ e'$, which binds x into e' but not e. This can be achieved generically as well. Suppose the binder $\lambda(x, e_1, e_2, \ldots, e_n)$ only binds x into $e_2, \ldots, e_n$, but not e_1, we can define the following syntactic sugar:

$$\lambda(x, e_1, \ldots, e_n) \equiv G\ \langle \mathit{int}\ \exists x.\langle x, e_2, \ldots, e_n\rangle, e_1\rangle$$

where all free occurrences of x in $e_2, \ldots, e_n$ are bound, but those in e_1 are still free.

We wrap up this section by leaving some open comments on the proof of Theorem 35, especially its completeness proof. As said, the proof goes through the models of λ-calculus, which is a complex subject that has attracted much attention since the proposal of λ-calculus and still has many unsolved problems [5]. However, it does not seem necessary to us that the proof needs to go through models. We conjecture that there exists a proof, which takes an *initial model* (see [23] for general notions on initial models; *term models* are special instances) of λ-calculus (modulo α- and β-equivalence), and turns it into an AML model. Then, if $e_1 = e_2$ holds in all AML models, it also holds in the one defined from the initial model, and thus it is provable, because the model is an initial one. We leave it as a future work to investigate this conjecture.

9 Instance: Type systems

In Sections 3, 6, 7 we showed that AML can define particular types and type relations and structures. To bring more support to the claim that AML can serve as a foundation for programming languages, in particular for defining type

$$\text{(Start)}\ \frac{\Gamma \vdash_{\mathsf{PTS}} A : s}{\Gamma, x{:}A \vdash_{\mathsf{PTS}} x{:}A} \qquad \text{(Axiom)}\ \ \emptyset \vdash_{\mathsf{PTS}} s_1 : s_2 \ \text{ if } (s_1, s_2) \in A$$

$$\text{(}\Pi\text{)}\ \frac{\Gamma \vdash_{\mathsf{PTS}} A : s_1 \quad \Gamma, x{:}A \vdash_{\mathsf{PTS}} B : s_2}{\Gamma \vdash_{\mathsf{PTS}} \Pi x{:}A.B : s_3} \ \text{ if } (s_1, s_2, s_3) \in R$$

$$\text{(}\Pi\text{-I)}\ \frac{\Gamma \vdash_{\mathsf{PTS}} \Pi x{:}A.B : s \quad \Gamma, x{:}A \vdash_{\mathsf{PTS}} b : B}{\Gamma \vdash_{\mathsf{PTS}} \lambda x{:}A.b : \Pi x{:}A.B} \qquad \text{(}\Pi\text{-E)}\ \frac{\Gamma \vdash_{\mathsf{PTS}} a : \Pi x{:}B.A \quad \Gamma \vdash_{\mathsf{PTS}} b : B}{\Gamma \vdash ab : A[b/x]}$$

$$\text{(Weak)}\ \frac{\Gamma \vdash_{\mathsf{PTS}} b : B \quad \Gamma \vdash_{\mathsf{PTS}} A : s}{\Gamma, x{:}A \vdash_{\mathsf{PTS}} b : B} \qquad \text{(Conv)}\ \frac{\Gamma \vdash_{\mathsf{PTS}} a : A \quad \Gamma \vdash_{\mathsf{PTS}} B : s \quad \vdash_\beta A = B}{\Gamma \vdash_{\mathsf{PTS}} a : B}$$

Figure 5: Typing rules of pure type system (here $\emptyset$ denotes the empty context)

systems for them, here we show how to define *pure type systems* [2, 59, 21] (shortened as PTS) as a theory in AML. PTS gives a uniform and simple approach to dependently-typed λ-calculus, λ-cubes [3], and type systems such as the Martin-Löf type system [36] and the calculus of constructions [15] (shortened as CoC), which is the foundation of proof assistants such as Coq [6], Agda [42], and Idris [9]. Appendix L shows that the technique below extends to the Martin-Löf type system.

Definition 36 ([21, Section 4.2])**.** Let Var be a set of *variables*, S be a set of *sorts*, and T be a set of *terms* defined as:
T (terms) $\quad a, b, A, B \coloneqq x \in \text{Var} \mid s \in S \mid ab \mid \lambda x{:}A.b \mid \Pi x{:}A.B$
As a convention, we use A, B, etc. to denote terms representing *types* and a, b, etc. to denote *data*. Here, λ and Π are *binders*, binding x into b and B, respectively, but not A. Thanks to α-renaming, we assume $x \notin \text{FV}(A)$ in $\lambda x{:}A.b$ and $\Pi x{:}A.B$. We modify (β) to the following typed version:

$$(\beta) \quad (\lambda x{:}A.b)\ a = b[a/x] \qquad \text{for all } x \in \text{Var} \text{ and } a, b, A \in T$$

where no type-checking happens yet. A *pure type system* is a pair (A, R) with a set $A \subseteq S \times S$ of *axioms* and a set $R \subseteq S \times S \times S$ of *rules*. A *typing context* is a sequence $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$ for $n \geq 0$ where all x_i's are distinct. The *typing relation*, $\Gamma \vdash_{\mathsf{PTS}} a{:}A$, is inductively defined in Fig. 5.

We show how to define PTS as an AML theory Γ^{PTS}, by modifying/extending Γ^{λ}, the theory of λ-calculus defined in Section 8, with the additional term constructors. Firstly, we define $\lambda x{:}A.b$ and $\Pi x{:}A.B$ as binders that bind x into b and B, but not A, using the generic method discussed in Section 8.5. Then, we interpret the inhabitant pattern $[\![A]\!]$ as one that is matched by all terms a such that a has type A, i.e., we define *typing* as syntactic sugar as follows:

$$a{:}A \equiv a \in [\![A]\!] \qquad \text{for all } a, A \in T$$

Next, we let Γ^{PTS} contain (β), the typed version, plus the following self-explanatory typing axioms:

$$\text{(Axiom)}\ \ s_1 : s_2 \ \text{ for } (s_1, s_2) \in A \qquad \text{(}\Pi\text{)}\ \ A : s_1 \wedge (\forall x{:}A.(B : s_2)) \to (\Pi x{:}A.B) : s_3 \ \text{ for } (s_1, s_2, s_3) \in R$$

$$\text{(}\Pi\text{-I)}\ \ (\Pi x{:}A.B) : s \wedge (\forall x{:}A.(b : B)) \to (\lambda x{:}A.b) : (\Pi x{:}A.B) \quad \text{(}\Pi\text{-E)}\ \ a : (\Pi x{:}B.A) \wedge b : B \to ab : A[b/x]$$

These axioms capture the corresponding PTS typing rules. Note that Γ was not needed and neither were axioms to capture (Start), (Weak), and (Conv); these follow from generic AML reasoning.

Theorem 37. *If* $x_1{:}A_1, \ldots, x_n{:}A_n \vdash_{\mathsf{PTS}} a{:}A$ *then* $\Gamma^{\mathsf{PTS}} \vdash x_1{:}A_1 \wedge \cdots \wedge x_n{:}A_n \to a{:}A$.

10 Application: Logic Foundation for Language Framework $\mathbb{K}$

Here we propose a semantic foundation of $\mathbb{K}$ as *rewriting modulo contexts*, completely within AML.

10.1 $\mathbb{K}$ preliminaries

$\mathbb{K}$ is a rewriting-based executable semantics framework for programming languages. It uses a succinct and modular meta-language to define both syntax and semantics of languages. As a running example, Fig. 6 shows the complete

```
1  module IMP-SYNTAX
2  imports DOMAINS-SYNTAX
3  syntax Exp ::=
4      Int
5    | Id
6    | Exp "+" Exp    [left, strict]
7    | Exp "-" Exp    [left, strict]
8    | "(" Exp ")"    [bracket]
9  syntax Stmt ::=
10     Id "=" Exp ";" [strict(2)]
11   | "if" "(" Exp ")"
12       Stmt Stmt    [strict(1)]
13   | "while" "(" Exp ")" Stmt
14   | "{" Stmt "}"   [bracket]
15   | "{" "}"
16   > Stmt Stmt   [left, strict(1)]
17 syntax Pgm ::= "int" Ids ";" Stmt
18 syntax Ids ::= List{Id,","}
19 endmodule
```

```
1  module IMP imports IMP-SYNTAX
2  imports DOMAINS
3  syntax KResult ::= Int
4  configuration <T> <k> $PGM:Pgm </k>
5                    <state> .Map </state> </T>
6  rule <k> X:Id => I ...</k>
7       <store>... X |-> I ...</store>
8  rule I1 + I2 => I1 +Int I2
9  rule I1 - I2 => I1 -Int I2
10 rule <k> X = I:Int => I ...</k>
11      <store>... X |-> (_ => I) ...</store>
12 rule {} S:Stmt => S
13 rule if(I) S _ => S requires I =/=Int 0
14 rule if(0) _ S => S
15 rule while(B) S => if(B) {S while(B) S} {}
16 rule <k> int (X, Xs => Xs) ; S </k>
17      <state>... (. => X |-> 0) </state>
18 rule int .Ids ; S => S
19 endmodule
```

Figure 6: The complete $\mathbb{K}$ definition of IMP, consisting of a syntax module and a semantic module

definition of IMP, a simple imperative language. IMP has common statements such as assignments, `if`-statements, `while`-loops, and sequential composition. The `IMP-SYNTAX` module defines its syntax and the `IMP` module defines its semantics. See `http://kframework.org` for details and papers on $\mathbb{K}$; we only give a high-level overview here.

Syntax

The syntax of IMP is defined using conventional BNF, where terminals are in quotes and nonterminal are not. Therefore, `Exp`, `Stmt`, `Pgm`, `Ids` are all nonterminals. `Int` and `Id` are also nonterminals, which are pre-defined and imported from `DOMAINS-SYNTAX`. Productions are separated by "|", or ">" (left, line 16) that states that subsequent productions have lower precedence.

Attributes

Productions can have attributes, listead in square brackets `[]`. Some attributes are purely for parsing purposes. For example, `left` (left, lines 6,7,16) indicates left-associativity; `bracket` (left, line 8) indicates that parentheses are for grouping, so the parser should not generate an AST node. Most attributes carry semantic meaning. For example, `strict`, `strict(1)`, etc., define *evaluation contexts* that affect how $\mathbb{K}$ executes programs. `strict` indicates that all arguments are evaluated first, so $e_1 + e_2$ is evaluated by first evaluating e_1 and e_2 to integers i_1 and i_2, *fully nondeterministicly*, and then evaluating $i_1 + i_2$. `strict(1)` indicates that only the first argument is evaluated, so `if(`b`)` P Q is evaluated by first evaluating b while *freezing* P and Q, as expected. The special nonterminal `KResult` (right, line 3) is used to give $\mathbb{K}$ hints when to stop searching for evaluation steps.

Attributes can also define *bindings*. For example, the λ-calculus syntax is defined as follows:

```
syntax Term ::= Id | "lambda" Id "." Term [binder] | Term Term [left]
```

where `Id` is a pre-defined sort for identifiers. The attribute `binder` specifies that `lambda` binds its first argument into the second, so internally, `lambda` is not defined as a constant symbol like other constructs, but as syntactic sugar following the generic method for defining binders in Section 8.5.

Configurations

$\mathbb{K}$ uses *configurations* (right, line 4) to store information needed for program execution. Configurations are organized into *cells*, which are labeled and can be nested. As a simple language, IMP has only three cells, but complex languages such as C can have hundreds [27]. The <k/> cell holds the *computation*, i.e., the program fragments that need to be

executed. The <state/> cell holds program states as finite maps from program variables to their values. Both cells are held in the top cell <T/>. Initially, the <k/> cell holds a program $PGM:Pgm, passed from the command-line, and the <state/> cell holds the empty map, denoted .Map.

Semantics

$\mathbb{K}$ defines semantics as *transition systems* over configurations in terms of a set of *rewrite rules* $\varphi_1 \Rightarrow \varphi_2$, where φ_1, φ_2 are configuration patterns. However, $\mathbb{K}$ introduces notations to write these rules compactly and modularly. For example, we do not need to mention full configurations in rewrite rules. We can only specify what matters and $\mathbb{K}$ infers the rest; e.g., instead of writing

```
rule <T> <k> I1 + I2 => I1 +Int I2 ...  </k> <state> M </state> </T>
```

("..." means evaluation context, explained in Section 10.2.3), we can write it more compactly as

```
rule I1 + I2 => I1 +Int I2         // as in module IMP, line 8
```

The latter rule tells $\mathbb{K}$ to rewrite I1 + I2 to I1 +Int I2 in *all contexts*, which include both evaluation contexts defined by the strictness attributes and configuration contexts defined by the cells. This makes $\mathbb{K}$ definitions modular and extensible, since rules need not change as configurations change.

Another useful $\mathbb{K}$ notation is the *local rewrite*, where the rewrite symbol => does not need to be at the top, and can appear locally at where the rewrite happens. So instead of writing

```
rule <k> X:Id ...</k> <state>...  X |-> I ...</state>
  => <k> I    ...</k> <state>...  X |-> I ...</state>
```

we can remove all the duplicate information and write it more compactly as

```
rule <k> X:Id => I ...</k> <state>...  X |-> I ...</state>      // as in module IMP, line 6
```

Program verification

Besides parsers and interpreters, $\mathbb{K}$ can generate several other language tools from a formal semantics. In particular, *program verification* is fully automatic in $\mathbb{K}$. That is, once given a program and a set of target (reachability) properties (including necessary invariant conditions), $\mathbb{K}$ carries out automatic reasoning in *reachability logic* [50] (RL), by symbolically executing the program and calling SMT solvers such as Z3 [18]. Recall Fig. 2, which shows that RL can be subsumed in AML as a theory, say Γ^{RL}. Then, program verification (in $\mathbb{K}$) is a best-effort implementation of automatic AML reasoning within Γ^{RL}.

10.2 Defining $\mathbb{K}$ semantics in applicative matching logic

As seen, $\mathbb{K}$ has an expressive front-end language to write compact, modular and extensible semantic definitions. Tools are then automatically generated from such $\mathbb{K}$ definitions. Here we show how to regard the $\mathbb{K}$ definitions as sugar for AML theories. Specifically, all the aforementioned features, from carrying out program execution and verification, to defining evaluation contexts, and to compact notations such as configuration inference and local rewrites, can be justified as AML notations and reasoning. We identify five research questions: (RQ1) how to define *static structures*, such as expressions, programs, cells, and configurations; (RQ2) how to define *dynamic transition relations* over configurations; (RQ3) how to define *evaluation contexts* generated by the strictness attributes; (RQ4) how to define *configuration inference* and (RQ5) how to define *local rewrites*.

10.2.1 Defining static program structures as patterns (RQ1)

Static structures can be defined as we define constructors and term algebras (Section 5). Specifically, for every nonterminal, such as Int, Exp, etc., we define a corresponding *sort* in Γ^{IMP}. Every production, say syntax Exp ::= Exp "+" Exp,

defines a corresponding constructor $\_+\_$: $\texttt{Exp} \times \texttt{Exp} \to \texttt{Exp}$. Productions such as `syntax Exp ::= Int` define "subsort" axioms such as $[\![\texttt{Int}]\!] \subseteq [\![\texttt{Exp}]\!]$, like in Section 6. Similarly, every cell, say <state/>, defines a corresponding constructor <state>_</state>: $\texttt{Map} \to \texttt{StateCell}$, where `Map` is the (pre-defined) sort of finite maps and `StateCell` is the sort of the <state/> cells. As a convention, we let $\texttt{Cfg} \equiv \texttt{TCell}$ denote the sort of the <top/> cells, i.e., program configurations.

10.2.2 Defining transition relations as patterns (RQ2)

Following an approach similar to [10, Section IX], we define the transition relation over configurations by defining a constant $\bullet$, called *one-path next*, with the following axiom:

$$\text{(One-Path Next)} \qquad \bullet\, [\![\texttt{Cfg}]\!] \subseteq [\![\texttt{Cfg}]\!]$$

Suppose φ is a configuration pattern. Then intuitively, $\bullet\varphi$ is matched by all configurations that have a next configuration (w.r.t. the transition relation) that matches φ. In other words, $\bullet\varphi$ is matched by all *predecessor configurations* of those matching φ. This is illustrated below:

$$\begin{array}{ccccccc} \cdots & c & \Rightarrow & c' & \Rightarrow & c'' & \cdots \quad \text{// configurations} \\ & \bullet\bullet\varphi & & \bullet\varphi & & \varphi & \quad \text{// patterns} \end{array}$$

Then, we can define rewrite rules as patterns using the one-path next constant: $\varphi \Rightarrow \psi \equiv \varphi \to \bullet\psi$.

10.2.3 Defining evaluation contexts as patterns (RQ3)

As seen in Section 10.1, a rewrite rule can only be applied under appropriate evaluation contexts. For example, consider configuration:

```
<T> <k> if (X + 1) { X = 3 - 2; } {} </k> <state> X |-> 0 </state> </T>
```

Let φ_{cfg} denote it. What rules can apply to φ_{cfg}? Indeed, there are many program/configuration fragments that *seem* to be rewritable: e.g., $\texttt{X} + 1$ could be rewritten to $\texttt{0} + 1$ by the variable lookup rule (Fig. 6, right, line 6); and $3 - 2$ could be rewritten to 1 by the arithmetic rule (Fig. 6, right, line 9). Rewriting systems such as Maude [13] would indeed rewrite $3 - 2$ to 1, because they do not rewrite *modulo contexts* by default. In $\mathbb{K}$, however, only the former rewrite is correct.

For simplicity, let φ_{if} denote the `if`-statement in φ_{cfg}, and let s_1 denote its `then`-branch and s_2 denote its `else`-branch. Recall that the `if`-statement is strict on its first argument, so we can *pull out* $\texttt{X} + 1$ from φ_{if}, and obtain the pattern $\texttt{if}(\square)\ s_1\ s_2$ with a distinguished placeholder $\square$. Note that $\square$ is just a dummy variable. We do not want to confuse it with other variables, so we *close* it by writing $\gamma\square.\texttt{if}(\square)\ s_1\ s_2$, where γ is a binder, defined as λ, and call the result a *context*. Formally:

Definition 38. A *context* is a pattern of the form $\gamma\square.\varphi$, where γ is a binder. Note that we only assume the same style to define the binder notation as for λ (Section 8), but we do not assume β-reduction. We write $C\ \varphi$ as $C[\varphi]$. For clarity, we define a new sort *Cxt* and the following axiom:

$$\text{(Context)} \qquad \gamma\square.\varphi \in [\![\mathit{Cxt}]\!]$$

Contexts of different types are not needed here, but those can be defined equally elegantly.

Now we can define evaluation contexts and the evaluation strategies in $\mathbb{K}$ in terms of axioms about contexts. For example, we define the following axiom for the `if`-statement context:

$$\text{(If-Statement Context)} \quad \forall b{:}\texttt{Exp}.\forall s_1{:}\texttt{Stmt}.\forall s_2{:}\texttt{Stmt}.\texttt{if}(b)\ s_1\ s_2 = (\gamma\square.\texttt{if}(\square)\ s_1\ s_2)[b]$$

In $\mathbb{K}$, such equations are used in both directions. When used from left to right, called *heating*, *redex* b is pulled out from its context. Where used from right to left, called *cooling*, b is plugged back into its context. The special sort `KResult` hints $\mathbb{K}$ to do heating/cooling efficiently, based on b's sort.

Similarly, the strictness of $\_+\_$ in its second argument corresponds to

$$\text{(Add Context Right)} \quad \forall e_1{:}\texttt{Exp}.\forall e_2{:}\texttt{Exp}\ .\ e_1 + e_2 = (\gamma\square.e_1 + \square)[e_2]$$

Using the above axioms, we can infer the following, which corresponds to what $\mathbb{K}$ does internally:

$$\begin{aligned}
\texttt{if(1+X)}\ s_1\ s_2 &= (\gamma\Box.\texttt{if(}\Box\texttt{)}\ s_1\ s_2)[\texttt{1+X}] && \text{// by (If-Statement Context)}\\
&= (\gamma\Box.\texttt{if(}\Box\texttt{)}\ s_1\ s_2)[(\gamma\Box.\texttt{1}+\Box)[\texttt{X}]] && \text{// by (Add Context Right)}
\end{aligned}$$

Now we encounter a nested context, which can be composed by the following generic axiom:

$$\text{(Composing Nested Contexts)}\quad \forall c_1 : Cxt\,.\,\forall c_2 : Cxt\,.\,c_1[c_2[x]] = (\gamma\Box\,.\,c_1[c_2[\Box]])[x]$$

Using this axiom, we continue the above reasoning, which also corresponds to what $\mathbb{K}$ does:

$$\begin{aligned}
\texttt{if(1+X)}\ s_1\ s_2 &= (\gamma\Box.\texttt{if(}\Box\texttt{)}\ s_1\ s_2)[(\gamma\Box.\texttt{1}+\Box)[\texttt{X}]] && \text{// already proved}\\
&= (\gamma\Box.(\gamma\Box.\texttt{if(}\Box\texttt{)}\ s_1\ s_2)[(\gamma\Box.\texttt{1}+\Box)[\Box]])[\texttt{X}] && \text{// composing nested contexts}\\
&= (\gamma\Box_1.(\gamma\Box_2.\texttt{if(}\Box_2\texttt{)}\ s_1\ s_2)[(\gamma\Box_3.\texttt{1}+\Box_3)[\Box_1]])[\texttt{X}] && \text{// } \alpha\text{-renaming}\\
&= (\gamma\Box_1.(\gamma\Box_2.\texttt{if(}\Box_2\texttt{)}\ s_1\ s_2)[\texttt{1}+\Box_1])[\texttt{X}] && \text{// by (Add Context Right)}\\
&= (\gamma\Box_1.(\texttt{if(1+}\Box_1\texttt{)}\ s_1\ s_2))[\texttt{X}] && \text{// by (If-Statement Context)}
\end{aligned}$$

Therefore, $\texttt{if(1+X)}\ s_1\ s_2$ is matched by a pattern $c[\texttt{X}]$, where c, as expected, is $\gamma\Box.(\texttt{if(1+}\Box\texttt{)}\ s_1\ s_2)$.

To sum up, "evaluation strategies" are AML reasoning using equations corresponding to contexts.

10.2.4 Defining configuration inference (RQ4)

Configuration inference is handled exactly the same way as the evaluation contexts described in Section 10.2.3. Configuration cells are symbols, same like the language syntax, which are automatically regarded as strict in all their arguments.

10.2.5 Defining local rewrites

In a local rewrite under context c, denoted $c[\varphi_1 \Rightarrow \varphi_2]$, the patterns φ_1, φ_2 might not be configuration patterns. Therefore, we first extend/overload $\bullet$ to all sorts s:

$$\text{(One-Path Next, Extended)}\qquad \bullet[\![s]\!] \subseteq [\![s]\!]$$

Then, we add a generic axiom that allows us to *lift* local rewrites to global rewrites:

$$\text{(Lift Rewrites)}\qquad \forall c : Cxt\,.\,c[\bullet\varphi] \rightarrow \bullet c[\varphi]$$

Finally, the following proposition shows that local rewrites imply global rewrites, if the context c is injective and extensional. Note that all program constructs and cells are defined as constructors, which are injective and extensional, so the following proposition holds for all implicit contexts.

Proposition 39. *Given a theory* Γ *and an* injective *and* extensional *context* c*, i.e.,* $\Gamma \vdash c[x] \wedge c[y] = c[x \wedge y]$*,* $\Gamma \vdash c[\bot] = \bot$ *and* $\Gamma \vdash c[x \vee y] = c[x] \vee c[y]$*, then* $\Gamma \vdash c[x \Rightarrow y]$ *implies* $\Gamma \vdash c[x] \Rightarrow c[y]$*.*

11 Proof checker for applicative matching logic

As seen, a number of important logical frameworks, illustrated in Fig. 2, are subsumed by AML as notations and/or theories. As a result, we can reason about these logical frameworks in a uniform way, using *one* fixed Hilbert-style proof system of AML (shown in Fig. 3), which allows us to prove sentences/judgments of the form $\Gamma \vdash \varphi$. As discussed in Sections 1&10, language frameworks (such as $\mathbb{K}$) are best effort *proof searchers* for $\Gamma \vdash \varphi$, with each tool optimized to support specific Γ and φ.

However, proof searching is complex and tools use heuristics and decision procedures that may contain bugs. For example, $\mathbb{K}$ has more than 130,000 LOC across several languages (including ¿60,000 Java, ¿35,000 Haskell, ¿34,000 OCaml). This places a huge "trustbase bet" for the users of such frameworks: they need to trust the *entire codespace*, even knowing it surely contains bugs.

	sorts/symbols	infrastructure	proof checking	total
size	46	35	22	103

Table 1: The total size of the proof checker of AML is ˜100 Maude statements / LOC; here, "sorts/symbols" denotes sort/symbol declarations that define the syntax of patterns as well as proof objects; "infrastructure" denotes equations defining substitution, α-renaming, and proof objects. Appendix N shows all the code.

Proof checkers are the key to *reducing trustbase* of language frameworks. Our preliminary experiments show that it reduces the trustbase of $\mathbb{K}$ from its ¿130,000-LOC codespace to a simple AML proof checker implemented in Maude [13] with ˜100 statements. Hilbert-style proof checking is very (!) simple. If $\Gamma \vdash \varphi$ is indeed a valid Hilbert-style proof, then there exists a sequence of patterns $\varphi_1, \ldots, \varphi_n$ such that $\varphi_n \equiv \varphi$ and every φ_i for $i \leq n$ is either an axiom in Γ or the result of applying an AML proof rule in Fig. 3. We can then put these information together as a *proof object*, which contains the sequence $\varphi_1, \ldots, \varphi_n$ as well as some *proof annotations*, specifying how each φ_i is derived. Obviously, a proof object is a much larger artifact than the original sentence/judgment $\Gamma \vdash \varphi$, but it contains all the details and thus can be easily 3$^{\text{rd}}$-party checked using proof checkers.

Appendix N shows the entire code of our AML proof checker. It defines the syntax of AML, substitution and α-renaming, and the core proof checking algorithm which follows blindly the proof system in Fig. 3. Table 1 shows some statistics about the proof checker.

12 Related and future work

We review a few related language frameworks here, and refer the reader to [64] for a survey of earlier frameworks. *CENTAUR* [8] is one of the earliest systems that take formal language definitions and automatically generate so-called *programming environments*, consisting of language tools such as interpreters and debuggers, equipped with graphic interfaces.

Proof assistants such as Coq [6] and Isabelle [41] have been extensively used as language frameworks, where program verification tasks or meta-theorems about languages are framed as theorems which are then carried out mostly interactively, sometimes requiring remarkable human effort, although (semi-)automation is available in some cases. Since defining a real language in a proof assistant is a tedious and thus error prone task, light-weight tools such as *Ott* [54] have been developed, which provide an intuitive front-end language to define formal syntax and semantics, accompanied with automatic tools that sanity-check definitions and translate them to proof assistants, where proofs are then carried out.

Besides $\mathbb{K}$, there are several other rewriting-based language frameworks. *Component-based specification (CBS)* [60] builds upon the observation that languages share many fundamental constructs, called *funcons*. CBS defines a rich set of funcons and uses them to define languages modularly. *Spoofax* [61] is a platform for designing *domain specific languages* (DSL), integrated with various language tools such as SDF [62] for formal syntax, Stratego [63] for code generation, FlowSpec [56] for data flow analysis, etc. *PLT Redex* [19] is a DSL for designing formal language semantics, fully integrated in a target programming language such as Scheme or Racket.

From a formal reasoning perspective, what is common to all language frameworks is that programming languages are defined as theories in an existing or hypothetical logic, and program execution and reasoning become logic deduction. We picked $\mathbb{K}$ as a case study not because we believe it is superior to any of the aforementioned frameworks, but because it is arguably one of the most complex and heavy on notation, so challenging to be given a semantics, and it is actively used to yield program verifiers for a variety of real languages, so it is in high need of a formal semantics.

One important direction for future work is an *interactive prover* for AML. Although language frameworks may provide automation for specific proof tasks, an exit solution is needed for cases when automation does not work. Like in other proof assistants, the interactive prover would explore the proof-space by executing user-defined or pre-defined proof strategies. Proof strategies can range from direct application of the AML proof rules to complex strategies employed by automated tools part of frameworks like $\mathbb{K}$ such as symbolic execution, pattern abstractions, and SMT-based domain reasoning. The latter would need to be enriched to produce proof objects, a well-known challenge in itself . To reuse the vast universe of formalized mathematics and well-engineered tooling in proof assistants like

Coq [6], Isabelle [41], Agda [42], etc., it would be useful to import and translate proof objects from these logical frameworks to proof objects in AML; preliminary work in Section 9 suggests that this is feasible.

13 Conclusion

We proposed a novel logic, called *applicative matching logic (AML)*, as a foundation for programming language frameworks. AML subsumes many logics/calculi/algebras important for programming languages, such as: FOL with least fixpoints, separation logic, temporal logics, modal μ-logic, reachability logic (which subsumes Hoare logic), many- and order-sorted algebras, term algebras, λ-calculi, and (dependent) type systems. We took as a case study the $\mathbb{K}$ framework, one of the most complex and actively used language frameworks, and showed how its main features are reduced to AML notations and reasoning. Finally, we also discussed a small checker for AML proof objects.

References

[1] Henk Barendregt. 1984. *The lambda calculus: Its syntax and semantics*. Elsevier Science Publishers.

[2] Henk Barendregt. 1991. Introduction to generalized type systems. *Journal of Functional Programming* 1, 2 (1991), 125–154. https://doi.org/10.1017/S0956796800020025

[3] Henk Barendregt. 1992. Lambda calculus with types. In *Handbook of logic in computer science*. Vol. 2. Oxford University Press, 117–309.

[4] Chantal Berline. 2000. From computation to foundations via functions and application: The λ-calculus and its webbed models. *Theoretical Computer Science* 249, 1 (2000), 81–161. https://doi.org/10.1016/S0304-3975(00)00057-8

[5] Chantal Berline. 2006. Graph models of λ-calculus at work, and variations. *Mathematical Structures in Computer Science* 16, 2 (2006), 185–221. https://doi.org/10.1017/S0960129506005123

[6] Yves Bertot and Pierre Castran. 2010. *Interactive theorem proving and program development: Coq'Art the calculus of inductive constructions*. Springer.

[7] Denis Bogdănaş and Grigore Roşu. 2015. K-Java: A complete semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, 445–456. https://doi.org/10.1145/2676726.2676982

[8] Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and V. Pascual. 1988. CENTAUR: The system. In *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE'88)*. ACM, 14–24. https://doi.org/10.1145/64135.65005

[9] Edwin Brady. 2011. IDRIS — Systems programming meets full dependent types. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification (PLPV '11)*. ACM, 43–54. https://doi.org/10.1145/1929529.1929536

[10] Xiaohong Chen and Grigore Roşu. 2019. Matching μ-logic. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19)*.

[11] Xiaohong Chen and Grigore Roşu. 2019. *Matching μ-logic*. Technical Report. University of Illinois at Urbana-Champaign. http://hdl.handle.net/2142/102281

[12] Alonzo Church. 1941. *The calculi of lambda-conversion*. Princeton University Press.

[13] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and José Quesada. 2002. Maude: specification and programming in rewriting logic. *Theoretical Computer Science* 285, 2 (2002), 187–243. https://doi.org/10.1016/S0304-3975(01)00359-0

[14] Anthony Cohn. 1989. Taxonomic reasoning with many-sorted logics. *Artificial Intelligence Review* 3, 2 (1989), 89–128. https://doi.org/10.1007/BF00128778

[15] Thierry Coquand and Gérard Huet. 1986. *The calculus of constructions*. Ph.D. Dissertation. INRIA.

[16] Haskell Curry. 1930. Grundlagen der kombinatorischen logik. *American Journal of Mathematics* 52, 3 (1930), 509–536. https://doi.org/10.2307/2370619

[17] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 2019. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. ACM. https://doi.org/10.1145/3314221.3314601

[18] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[19] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics engineering with PLT Redex*. Mit Press.

[20] Michael J. Fischer and Richard E. Ladner. 1979. Propositional dynamic logic of regular programs. *J. Comput. System Sci.* 18, 2 (1979), 194–211. https://doi.org/10.1016/0022-0000(79)90046-1

[21] Herman Geuvers. 1993. *Logics and type systems*.

[22] Joseph Goguen and José Meseguer. 1992. Order-sorted algebra, Part 1: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science* 105, 2 (1992), 217–273. https://doi.org/10.1016/0304-3975(92)90302-V

[23] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. 1977. Initial algebra semantics and continuous algebras. *Journal of the ACM* 24, 1 (1977), 68–95. https://doi.org/10.1145/321992.321997

[24] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. 2000. *Software engineering with OBJ: Algebraic specification in action*. Springer, Chapter Introducing OBJ, 3–167. https://doi.org/10.1007/978-1-4757-6541-0_1

[25] Yuri Gurevich and Saharon Shelah. 1985. Fixed-point extensions of first-order logic. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (SFCS'85)*. IEEE, 346–353. https://doi.org/10.1109/SFCS.1985.27

[26] David Harel. 1984. Dynamic logic. In *Handbook of Philosophical Logic*. Vol. 165. Springer, 497–604. https://doi.org/10.1007/978-94-009-6259-0

[27] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the undefinedness of C. In *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 336–345. https://doi.org/10.1145/2813885.2737979

[28] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2018. KEVM: A complete semantics of the Ethereum virtual machine. In *Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF'18)*. IEEE. http://jellopaper.org.

[29] Roger Hindley and Giuseppe Longo. 1980. Lambda-calculus models and extensionality. *Mathematical Logic Quarterly* 26, 4 (1980), 289–310. https://doi.org/10.1002/malq.19800261902

[30] Theodoros Kasampalis, Dwight Guth, Brandon Moore, Traian Florin Şerbănuţă, Yi Zhang, Daniele Filaretti, Virgil Şerbănuţă, Ralph Johnson, and Grigore Roşu. 2019. IELE: A rigorously designed language and tool ecosystem for the blockchain. In *Proceeding of the 23rd International Symposium on Formal Methods (FM'19)*.

[31] C. P. J. Koymans. 1982. Models of the lambda calculus. *Information and Control* 52 (1982), 306–332.

[32] Dexter Kozen. 1982. Results on the propositional μ-calculus. In *Proceedings of the 9th International Colloquium on Automata, Languages and Programming (ICALP'82)*. Springer, 348–359. https://doi.org/10.1007/BFb0012782

[33] Liyi Li and Elsa Gunter. 2018. IsaK-static A complete static semantics of K. In *Formal Aspects of Component Software*. Springer, 196–215. https://doi.org/10.1007/978-3-030-02146-7_10

[34] Anatoli Ivanovi Malc'ev. 1936. Axiomatizable classes of locally free algebras of various type. *The Metamathematics of Algebraic Systems: Collected Papers* (1936), 262–281.

[35] Giulio Manzonetto. 2008. *Models and theories of lambda calculus*. Ph.D. Dissertation. Università Ca' Foscari di Venezia. https://tel.archives-ouvertes.fr/tel-00715207

[36] Per Martin-Löf. 1975. An intuitionistic theory of types: Predicative part. In *Logic Colloquium*. Vol. 80. Elsevier, 73–118. https://doi.org/10.1016/S0049-237X(08)71945-1

[37] Elliott Mendelson. 1979. *Introduction to mathematical logic*. Springer.

[38] José Meseguer and Joseph Goguen. 1993. Order-sorted algebra solves the constructor-selector, multiple representation, and coercion problems. *Information and Computation* 103, 1 (1993), 114–158. https://doi.org/10.1006/inco.1993.1016

[39] Brandon Moore, Lucas Peña, and Grigore Roşu. 2018. Program verification by coinduction. In *Proceedings of the 27th European Symposium on Programming (ESOP'18)*. Springer, 589–618. https://doi.org/10.1007/978-3-319-89884-1_21

[40] Timothy Nelson, Daniel Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2010. *On the finite model property in order-sorted logic*. Technical Report. Worcester Polytechnic Institute.

[41] Tobias Nipkow, Markus Wenzel, and Lawrence Paulson. 2002. *Isabelle/HOL: A proof assistant for higher-order logic*. Springer.

[42] Ulf Norell. 2009. Dependently typed programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming (AFP'09)*. Springer, 230–266. https://doi.org/10.1007/978-3-642-04652-0_5

[43] Peter O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. https://doi.org/10.1145/3211968

[44] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2015. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 346–356. https://doi.org/10.1145/2737924.2737991

[45] Giuseppe Peano. 1889. *Arithmetices principia: Nova methodo exposita*. Fratres Bocca.

[46] Amir Pnueli. 1977. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS'77)*. IEEE, 46–57. https://doi.org/10.1109/SFCS.1977.32

[47] John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*. IEEE, 55–74. https://doi.org/10.1109/lics.2002.1029817

[48] Grigore Roşu. 2017. Matching logic. *Logical Methods in Computer Science* 13, 4 (2017), 1–61. https://doi.org/10.23638/lmcs-13(4:28)2017

[49] Grigore Roşu and Andrei Ştefănescu. 2012. From Hoare logic to matching logic reachability. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*. Springer, 387–402. https://doi.org/10.1007/978-3-642-32759-9_32

[50] Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. 2013. One-path reachability logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*. IEEE, 358–367. https://doi.org/10.1109/lics.2013.42

[51] John Barkley Rosser. 1935. A mathematical logic without variables, Part 1. *Annals of Mathematics* 36, 1 (1935), 127–150. https://doi.org/10.2307/1968669

[52] Moses Schönfinkel. 1924. Über die bausteine der mathematischen logik. *Mathematische annalen* 92, 3-4 (1924), 305–316. https://doi.org/10.1007/BF01448013

[53] Traian Florin Şerbănuţă and Grigore Roşu. 2012. A truly concurrent semantics for the K framework based on graph transformations. In *Proceedings of the 6th International Conference on Graph Transformation (ICGT'12)*. Springer, 294–310. https://doi.org/10.1007/978-3-642-33654-6_20

[54] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2010. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming* 20, 1 (2010), 71–122. https://doi.org/10.1017/S0956796809990293

[55] Joseph R. Shoenfield. 1967. *Mathematical logic*. Addison-Wesley Pub. Co. https://doi.org/10.1201/9780203749456

[56] Jeff Smits and Eelco Visser. 2017. FlowSpec: Declarative dataflow analysis specification. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE'17)*. ACM, 221–231. https://doi.org/10.1145/3136014.3136029

[57] Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (1955), 285–309. https://doi.org/10.2140/pjm.1955.5.285

[58] Anne Sjerp Troelstra. 1987. On the syntax of Martin-Löf's type theories. *Theoretical Computer Science* 51, 1 (1987), 1–26. https://doi.org/10.1016/0304-3975(87)90047-8

[59] L. S. van Benthem Jutting. 1993. Typing in pure type systems. *Information and Computation* 105, 1 (1993), 30–41. https://doi.org/10.1006/inco.1993.1038

[60] L. Thomas van Binsbergen, Neil Sculthorpe, and Peter D. Mosses. 2016. Tool support for component-based semantics. In *Companion Proceedings of the 15th International Conference on Modularity*. ACM, 8–11. https://doi.org/10.1145/2892664.2893464

[61] Mark van den Brand, Jan Heering, Paul Klint, and Pieter A. Olivier. 2002. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS'02)* 24, 4 (2002), 334–368. https://doi.org/10.1145/567097.567099

[62] Eelco Visser. 1997. *Syntax definition for language prototyping*. Ph.D. Dissertation. University of Amsterdam.

[63] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. 1998. Building program optimizers with rewriting strategies. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*. ACM, 13–26. https://doi.org/10.1145/289423.289425

[64] Yingzhou Zhang and Baowen Xu. 2004. A survey of semantic description frameworks for programming languages. *ACM SIGPLAN Notices* 39, 3 (2004), 14–30. https://doi.org/10.1145/981009.981013

A Proofs about basic definitions and notations

Here we prove all propositions and theorems in Section 2.

Proposition 5. *Under the above notation, the following hold:*

$$\begin{array}{lll} \bar{\rho}(\neg\varphi) = M \setminus \bar{\rho}(\varphi) & \bar{\rho}(\varphi_1 \vee \varphi_2) = \bar{\rho}(\varphi_1) \cup \bar{\rho}(\varphi_2) & \bar{\rho}(\varphi_1 \wedge \varphi_2) = \bar{\rho}(\varphi_1) \cap \bar{\rho}(\varphi_2) \\ \bar{\rho}(\top) = M & \bar{\rho}(\varphi_1 \leftrightarrow \varphi_2) = M \setminus (\bar{\rho}(\varphi_1) \triangle \bar{\rho}(\varphi_2)) & \bar{\rho}(\forall x.\varphi) = \bigcap_{a\in M} \overline{\rho[a/x]}(\varphi) \\ \bar{\rho}(\nu X.\varphi) = \nu\mathcal{F}^{\rho}_{\varphi,X} & \text{with } \mathcal{F}^{\rho}_{\varphi,X} \text{ defined the same as in Definition 4} & \end{array}$$

where "$\triangle$" denotes set symmetric difference: $A \triangle B = (A \setminus B) \cup (B \setminus A)$.

Proof. Simple, by applying definitions directly.

(Case $\neg\varphi$): $\bar{\rho}(\neg\varphi) = \bar{\rho}(\varphi \to \bot) = M \setminus (\bar{\rho}(\varphi) \setminus \bar{\rho}(\bot)) = M \setminus (\bar{\rho}(\varphi) \setminus \emptyset) = M \setminus \bar{\rho}(\varphi)$.

(Case $\varphi_1 \vee \varphi_2$): $\bar{\rho}(\varphi_1 \vee \varphi_2) = \bar{\rho}(\neg\varphi_1 \to \varphi_2) = M \setminus (\bar{\rho}(\neg\varphi_1) \setminus \bar{\rho}(\varphi_2)) = M \setminus ((M \setminus \bar{\rho}(\varphi_1)) \setminus \bar{\rho}(\varphi_2)) = \bar{\rho}(\varphi_1) \cup \bar{\rho}(\varphi_2)$.

(Case $\varphi_1 \wedge \varphi_2$): $\bar{\rho}(\varphi_1 \wedge \varphi_2) = \bar{\rho}(\neg\varphi_1 \vee \neg\varphi_2) = \bar{\rho}(\neg\varphi_1) \vee \bar{\rho}(\neg\varphi_2) = (M \setminus \bar{\rho}(\varphi_1)) \vee (M \setminus \bar{\rho}(\varphi_2)) = \bar{\rho}(\varphi_1) \wedge \bar{\rho}(\varphi_2)$.

(Case $\top$): $\bar{\rho}(\top) = \bar{\rho}(\neg\bot) = M \setminus \bar{\rho}(\bot) = M \setminus \emptyset = M$.

(Case $\varphi_1 \leftrightarrow \varphi_2$): $\bar{\rho}(\varphi_1 \leftrightarrow \varphi_2) = \bar{\rho}((\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1)) = \bar{\rho}(\varphi_1 \to \varphi_2) \cap \bar{\rho}(\varphi_2 \to \varphi_1) = (M \setminus (\bar{\rho}(\varphi_1) \setminus \bar{\rho}(\varphi_2))) \cap (M \setminus (\bar{\rho}(\varphi_2) \setminus \bar{\rho}(\varphi_1))) = M \setminus (\bar{\rho}(\varphi_1) \triangle \bar{\rho}(\varphi_2))$.

(Case $\forall x.\varphi$): $\bar{\rho}(\forall x.\varphi) = \bar{\rho}(\neg\exists x.\neg\varphi) = M \setminus \bar{\rho}(\exists x.\neg\varphi) = M \setminus \bigcup_{a\in M} \overline{\rho[a/x]}(\neg\varphi) = M \setminus \bigcup_{a\in M} \left(M \setminus \overline{\rho[a/x]}(\varphi)\right) = \bigcap_{a\in M} \overline{\rho[a/x]}(\varphi)$.

(Case $\nu X.\varphi$): $\bar{\rho}(\nu X.\varphi) = \bar{\rho}(\neg\mu X.\neg\varphi[\neg X/X]) = M \setminus \bar{\rho}(\mu X.\neg\varphi[\neg X/X]) = M \setminus \mu\mathcal{F}^{\rho}_{\neg\varphi[\neg X/X],X}$. Note that $\mu\mathcal{F}^{\rho}_{\neg\varphi[\neg X/X]} = \bigcup\{A \subseteq M \mid \mathcal{F}^{\rho}_{\neg\varphi[\neg X/X]}(A) \subseteq A\} = \bigcup\{A \subseteq M \mid \overline{\rho[A/X]}(\neg\varphi[\neg X/X]) \subseteq A\}$. Then $M \setminus \mu\mathcal{F}^{\rho}_{\neg\varphi[\neg X/X],X} = M \setminus \bigcup\{A \subseteq M \mid \overline{\rho[A/X]}(\neg\varphi[\neg X/X])\} = \bigcap\{M \setminus A \mid \overline{\rho[A/X]}(\neg\varphi[\neg X/X]) \subseteq A\} = \bigcap\{B \mid \overline{\rho[B/X]}(\neg\varphi) \subseteq (M \setminus B)\} = \bigcap\{B \mid \overline{\rho[B/X]}(\varphi) \supseteq B\} = \nu\mathcal{F}^{\rho}_{\varphi,X}$ □

Proposition 8. *With the above notation, the following hold:*

- $\bar{\rho}(\lceil\varphi\rceil) = M$ *iff* $\bar{\rho}(\varphi) \neq \emptyset$*; and* $\bar{\rho}(\lceil\varphi\rceil) = \emptyset$ *iff* $\bar{\rho}(\varphi) = \emptyset$*;*
- $\bar{\rho}(\lfloor\varphi\rfloor) = M$ *iff* $\bar{\rho}(\varphi) = M$*; and* $\bar{\rho}(\lfloor\varphi\rfloor) = \emptyset$ *iff* $\bar{\rho}(\varphi) \neq \emptyset$*;*
- $\bar{\rho}(\varphi_1 = \varphi_2) = M$ *iff* $\bar{\rho}(\varphi_1) = \bar{\rho}(\varphi_2)$*; and* $\bar{\rho}(\varphi_1 = \varphi_2) = \emptyset$ *iff* $\bar{\rho}(\varphi_1) \neq \bar{\rho}(\varphi_2)$*;*
- $\bar{\rho}(x \in \varphi) = M$ *iff* $\rho(x) \in \bar{\rho}(\varphi)$*; and* $\bar{\rho}(x \in \varphi) = \emptyset$ *iff* $\rho(x) \notin \bar{\rho}(\varphi)$*;*
- $\bar{\rho}(\varphi_1 \subseteq \varphi_2) = M$ *iff* $\bar{\rho}(\varphi_1) \subseteq \bar{\rho}(\varphi_2)$*; and* $\bar{\rho}(\varphi_1 \subseteq \varphi_2) = \emptyset$ *iff* $\bar{\rho}(\varphi_1) \not\subseteq \bar{\rho}(\varphi_2)$*.*

Proof. These can be proved by simply applying the definitions.

(Case $\lceil\varphi\rceil$): $\bar{\rho}(\lceil\varphi\rceil) = M$ iff $\lceil\_\rceil_M \cdot \bar{\rho}(\varphi) = M$ iff there exists $a \in \bar{\rho}(\varphi)$ iff $\bar{\rho}(\varphi) \neq \emptyset$. Otherwise, $\bar{\rho}(\lceil\varphi\rceil) = \emptyset$ iff $\lceil\_\rceil_M \cdot \bar{\rho}(\varphi) = \emptyset$ iff there exists no $a \in \bar{\rho}(\varphi)$ iff $\bar{\rho}(\varphi) = \emptyset$.

(Case $\lfloor\varphi\rfloor$): $\bar{\rho}(\lfloor\varphi\rfloor) = M$ iff $\bar{\rho}(\neg\lceil\neg\varphi\rceil) = M$ iff $\bar{\rho}(\lceil\neg\varphi\rceil) = \emptyset$ iff $\bar{\rho}(\neg\varphi) = \emptyset$ iff $\bar{\rho}(\varphi) = M$. Otherwise, $\bar{\rho}(\lfloor\varphi\rfloor) = \emptyset$ iff $\bar{\rho}(\neg\lceil\neg\varphi\rceil) = \emptyset$ iff $\bar{\rho}(\lceil\neg\varphi\rceil) = M$ iff $\bar{\rho}(\neg\varphi) = M$ iff $\bar{\rho}(\varphi) = \emptyset$.

(Case $\varphi_1 = \varphi_2$): $\bar{\rho}(\varphi_1 = \varphi_2) = M$ iff $\bar{\rho}(\lfloor\varphi_1 \leftrightarrow \varphi_2\rfloor) = M$ iff $\bar{\rho}(\varphi_1 \leftrightarrow \varphi_2) = M$ iff $\bar{\rho}(\varphi_1) \triangle \bar{\rho}(\varphi_2) = \emptyset$ iff $\bar{\rho}(\varphi_1) = \bar{\rho}(\varphi_2)$. Otherwise, $\bar{\rho}(\varphi_1 = \varphi_2) = \emptyset$ iff $\bar{\rho}(\lfloor\varphi_1 \leftrightarrow \varphi_2\rfloor) = \emptyset$ iff $\bar{\rho}(\varphi_1 \leftrightarrow \varphi_2) = \emptyset$ iff $\bar{\rho}(\varphi_1) \triangle \bar{\rho}(\varphi_2) = M$ iff $\bar{\rho}(\varphi_1) \neq \bar{\rho}(\varphi_2)$.

(Case $x \in \varphi$): $\bar{\rho}(x \in \varphi) = M$ iff $\bar{\rho}(\lceil x \wedge \varphi\rceil) = M$ iff $\bar{\rho}(x \wedge \varphi) \neq \emptyset$ iff $\{\rho(x)\} \cap \bar{\rho}(\varphi) \neq \emptyset$ iff $\rho(x) \in \bar{\rho}(\varphi)$. Otherwise, $\bar{\rho}(x \in \varphi) = \emptyset$ iff $\bar{\rho}(\lceil x \wedge \varphi\rceil) = \emptyset$ iff $\bar{\rho}(x \wedge \varphi) = \emptyset$ iff $\{\rho(x)\} \cap \bar{\rho}(\varphi) = \emptyset$ iff $\rho(x) \notin \bar{\rho}(\varphi)$.

(Case $\varphi_1 \subseteq \varphi_2$): $\bar{\rho}(\varphi_1 \subseteq \varphi_2) = M$ iff $\bar{\rho}(\lfloor\varphi_1 \to \varphi_2\rfloor) = M$ iff $\bar{\rho}(\varphi_1 \to \varphi_2) = M$ iff $M \setminus (\bar{\rho}(\varphi_1) \setminus \bar{\rho}(\varphi_2)) = M$ iff $\bar{\rho}(\varphi_1) \setminus \bar{\rho}(\varphi_2) = \emptyset$ iff $\bar{\rho}(\varphi_1) \subseteq \bar{\rho}(\varphi_2)$. Otherwise, $\bar{\rho}(\varphi_1 \subseteq \varphi_2) = \emptyset$ iff $\bar{\rho}(\lfloor\varphi_1 \to \varphi_2\rfloor) = \emptyset$ iff $\bar{\rho}(\varphi_1 \to \varphi_2) \neq M$ iff $M \setminus (\bar{\rho}(\varphi_1) \setminus \bar{\rho}(\varphi_2)) \neq M$ iff $\bar{\rho}(\varphi_1) \setminus \bar{\rho}(\varphi_2) \neq \emptyset$ iff $\bar{\rho}(\varphi_1) \not\subseteq \bar{\rho}(\varphi_2)$. □

Proposition 9. *Let σ be any constant symbol. We define the following two axioms:*

$$\text{(Functional Constant)}\ \exists z.\sigma = z \qquad \text{(Functional Application)}\ \exists z.xy = z$$

Then, for any model M satisfying (Functional Constant)*, σ_M is a singleton set, and for any model M satisfying* (Functional Application)*, its application is a function, i.e., $|a \cdot b| = 1$ for all $a, b \in M$.*

Proof. Suppose M satisfies (Functional Constant). Then for any ρ, $\bar{\rho}(\exists z.\sigma = z) = M$, i.e., $\bigcup_{c\in M} \overline{\rho[c/z]}(\sigma = z) = M$. Note that $\overline{\rho[c/z]}(\sigma = z)$ is either $\emptyset$ or M, so there exists $c \in M$ such that $\overline{\rho[c/z]}(\sigma = z)$, i.e., $\sigma_M = \{c\}$, which is a singleton.

Suppose M satisfies (Functional Application). Then for any ρ, $\bar{\rho}(\exists z.xy = z) = M$, i.e., $\bigcup_{c\in M} \overline{\rho[c/z]}(xy = z) = M$. There exists $c \in M$ such that $\overline{\rho[c/z]}(xy = z) = M$, i.e., $\rho(x) \cdot \rho(y) = \{c\}$. Note that ρ is arbitrary. Then for all $a, b \in M$ there exists $c \in M$ such that $a \cdot b = \{c\}$. □

We tacitly blur the distinction between elements and singleton sets.

Proposition 11. *Any AML model satisfying* (Functional Application) *is an applicative structure. Additionally, if the AML theory includes two functional constants $k, s \in \Sigma$ satisfying the axioms $kxy = x$ and $sxyz = xy(xz)$, then its models are combinatory algebras.*

Proof. By Proposition 9 and Definition 10. □

Proposition 13. *For all theories Γ with definedness, the following hold:*

$\Gamma \vdash \varphi = \varphi$ $\qquad$ $\Gamma \vdash \varphi_1 = \varphi_2$ *and* $\Gamma \vdash \varphi_2 = \varphi_3$ *implies* $\Gamma \vdash \varphi_1 = \varphi_3$

$\Gamma \vdash \varphi_1 = \varphi_2$ *implies* $\Gamma \vdash \varphi_2 = \varphi_2$ $\qquad$ $\Gamma \vdash \varphi_1 = \varphi_2$ *implies* $\Gamma \vdash \psi[\varphi_1/x] = \psi[\varphi_2/x]$.

Proof. See [11, Lemma 50 and 60]. □

Theorem 14 (Soundness Theorem)**.** $\Gamma \vdash \varphi$ *implies* $\Gamma \vDash \varphi$.

Proof. See [11, Theorem 13]. □

A.1 More about definedness

In Proposition 11, we give an axiomatic characterization of when AML models are applicative structures. Here we consider the other direction and show that all applicative structures are also AML models. Firstly, note the following property about definedness.

Proposition 40. *Let M be an AML model satisfying* (Definedness)*. Then $M \cdot a = M$ for all $a \in M$, i.e., for all $a, b \in M$ there exists $c \in M$ such that $b \in c \cdot a$.*

Proof. Let $\lceil \_ \rceil_M \subseteq M$ be the interpretation of definedness in M. Since M satisfies (Definedness), we have $\lceil \_ \rceil_M \cdot a = M$ for all $a \in M$. Then by pointwise extension, $M \cdot a = M$. □

When M has functional application, the conclusion in Proposition 40 becomes: for all $a, b \in M$ there exists $c \in M$ such that $b = c \cdot a$. If an applicative structure does not satisfies this property, it cannot give coherent interpretation for definedness, and thus is not an AML model. This difficulty can be solved by assuming a special element, say $\$ \in M$, such that $\$ \cdot a = M$ for all $a \in M$. Then, we use $\$ as the interpretation of definedness and use the restricted model, $M \setminus \{\$\}$, to capture applicative structures and combinatory algebras. In the following, we tacitly assume $\$ in all AML models and use it to interpret the definedness symbol, with the property that $\$ \cdot a = M$ for all $a \in M$.

B Proof of Theorem 17

Theorem 17. *(S, Σ)-MSA are exactly the restricted Γ^{MSA}-models w.r.t. (S, Σ).*

Proof. All restricted Γ^{MSA}-models w.r.t. (S, Σ) are (S, Σ)-MSA, by definition, so we just need to show the other direction. For any (S, Σ)-MSA, say $A = (\{A_s\}_{s\in S}, \{f_A\}_{f\in\Sigma})$, we need to find an AML model $M \vDash \Gamma^{\mathsf{MSA}}$, whose restricted model w.r.t. (S, Σ) is exactly A.

We first define the carrier set M. Let M contain/include all the following elements/sets:

- $\$ be a distinguished element denoting definedness; see Appendix A.1;

- # be a distinguished element denoting inhabitant; for the same reason why we have \$;
- S, the sort set;
- A_s, for all $s \in S$;
- $[A_{s_i} \to [A_{s_{i+1}} \to [\cdots \to [A_{s_n} \to A_s] \dots]]]$ for all $\Sigma_{s_1 \dots s_n, s} \neq \emptyset$ and $1 \leq i \leq n$. Here $[A \to B]$ denotes the set of all functions from A to B; intuitively, this is for the interpretation of functions and their partial application;

Next, define the following interpretations of constants:

- $\lceil \_ \rceil_M = \{\$\}$ and $[\![\_]\!]_M = \{\#\}$;
- $s_M = \{s\}$ for all $s \in S$;
- $f_M = \{f_A\}$ for all $f \in \Sigma_{s_1 \dots s_n, s}$; note that $f_A \colon M_{s_1} \times \cdots \times M_{s_n} \to M_s$, under the *curry-uncurry isomorphism*, is an element in $[A_{s_1} \to [A_{s_2} \to [\cdots \to [A_{s_n} \to A_s] \cdots]]]$. Therefore, $f_A \in M$ and f_M is well-defined.

Next, we define the application in M as the following:

- $\$ \bullet a = M$ for all $a \in M$;
- $\# \bullet s = A_s$ for all $s \in S$; note that $[\![s_M]\!]_M = \# \bullet \{s\} = A_s$;
- $f \bullet a = \{f(a)\}$ for all $f \in [A \to B]$ and $a \in A$; that is, application is interpreted as the normal function application, if the first argument is a function and the second is an element of the appropriate sort.

The unmentioned cases are irrelevant.

Now, we verify that M satisfies all axioms in Γ^{MSA}. (Definedness) is satisfied by definition. (Nonempty Sort) is satisfied because $A_s \neq \emptyset$ for all $s \in S$. (Function) for every $f \in \Sigma_{s_1 \dots s_n, s}$ is satisfied because f_M is defined by f_A and the application $\_ \bullet \_$ is interpreted as the normal function application. Therefore, $M \vDash \Gamma^{\mathsf{MSA}}$.

Finally, we prove that the restricted model M^r w.r.t. (S, Σ) is exactly A. By definition, $M^r = (\{M^r_s\}_{s \in S}, \{f_{M^r}\}_{f \in \Sigma})$ is an (S, Σ)-MSA defined as follows:

- the carrier set $M^r_s = [\![s_M]\!]_M$, which equals A_s as we showed above, for all $s \in S$;
- the interpretation f_{M^r}, for $f \in \Sigma_{s_1 \dots s_n, s}$, is defined such that $\{f_{M^r}(a_1, \dots, a_n)\} = f_M a_1 \cdots a_n$ for all $a_1 \in A_{s_1}, \dots, a_n \in A_{s_n}$; note that $f_M a_1 \cdots a_n = \{f_A\} a_1 \cdots a_n = \{f_A(a_1, \dots, a_n)\}$, so we have $f_{M^r}(a_1, \dots, a_n) = f_A(a_1, \dots, a_n)$, for all $a_1 \in A_{s_1}, \dots, a_n \in A_{s_n}$.

Therefore, M^r is exactly A. □

C Proof of Proposition 18

Proposition 18. *Under the above notations,* $\vdash \forall x{:}s \,.\, \varphi = \neg \exists x{:}s \,.\, \neg \varphi$ *and* $\vdash \exists x{:}s \,.\, \varphi = \neg \forall x{:}s \,.\, \neg \varphi$.

Proof. We prove the first one as follows:

$$
\begin{aligned}
\forall x{:}s \,.\, \varphi &= \forall x \,.\, x \in [\![s]\!] \to \varphi && \text{// by definition} \\
&= \neg \exists x \,.\, \neg (x \in [\![s]\!] \to \varphi) && \text{// by FOL reasoning} \\
&= \neg \exists x \,.\, x \in [\![s]\!] \wedge \neg \varphi && \text{// by FOL reasoning} \\
&= \neg \exists x{:}s \,.\, \varphi && \text{// by definition}
\end{aligned}
$$

The second can be proved in the same way. □

The following proposition is useful in reasoning about the semantics of sorted quantification.

Proposition 41. *Let M be a model, s be a sort, whose inhabitant set is denoted/defined as $M_s = [\![s_M]\!]_M$. Then for any valuation ρ, $\bar{\rho}(\exists x\colon s\,.\,\varphi) = \bigcup_{a\in M_s} \overline{\rho[a/x]}(\varphi)$ and $\bar{\rho}(\forall x\colon s\,.\,\varphi) = \bigcap_{a\in M_s} \overline{\rho[a/x]}(\varphi)$*

Proof. We prove for $\exists x\colon s\,.\,\varphi$. The proof of the other one is similar. By definition, $\bar{\rho}(\exists x\colon s\,.\,\varphi) = \bar{\rho}(\exists x\,.\,x \in [\![s]\!] \wedge \varphi) = \bigcup_{a\in M} \left(\overline{\rho[a/x]}(x \in [\![s]\!]) \cap \overline{\rho[a/x]}(\varphi)\right)$. Note that $\overline{\rho[a/x]}(x \in [\![s]\!]) = M$ iff $\rho[a/x](x) \in \overline{\rho[a/x]}([\![s]\!])$, i.e., $a \in M_s$; and $\overline{\rho[a/x]}(x \in [\![s]\!]) = \emptyset$ iff $a \notin M_s$; see Proposition 5. Therefore, $\bar{\rho}(\exists x\colon s\,.\,\varphi) = \bigcup_{a\in M_s} \left(\overline{\rho[a/x]}(x \in [\![s]\!]) \cap \overline{\rho[a/x]}(\varphi)\right) = \bigcup_{a\in M_s} \overline{\rho[a/x]}(\varphi)$. □

D Proofs about matching μ-logic

D.1 Proof of Theorem 21

Theorem 21. *(S, V, Σ)-models of MmL are exactly the restricted Γ^{MmL}-models w.r.t. (S, V, Σ).*

Proof. All restricted Γ^{MmL}-models w.r.t. (S, V, Σ) are (S, V, Σ)-models of MmL, by definition, so we just need to show the other direction. For any (S, Σ)-model, say $M = (\{M_s\}, \{\sigma\}_{\sigma\in\Sigma})$, we need to find an AML model $M^{\mathsf{AML}} \vDash \Gamma^{\mathsf{MmL}}$, whose restricted model w.r.t. (S, V, Σ), is exactly M.

We first define the carrier set M^{AML}, by letting it contain/include all the following elements/sets:

- \$ for definedness and # for inhabitant;
- S, the sort set;
- M_s for all $s \in S$;
- $[M_{s_i} \to [M_{s_{i+1}} \to [\cdots \to [M_{s_n} \to \mathcal{P}(M_s)] \ldots]]]$ for all $\Sigma_{s_1\ldots s_n,s} \neq \emptyset$ and $1 \leq i \leq n$; this is for interpreting symbols and their partial applications; note that powerset semantics of MmL is captured by the co-domain $\mathcal{P}(M_s)$.

Next, we define the following interpretations of constants:

- $\lceil\_\rceil_{M^{\mathsf{AML}}} = \{\$\}$ and $[\![\_]\!]_{M^{\mathsf{AML}}} = \{\#\}$;
- $s_{M^{\mathsf{AML}}} = \{s\}$ for all $s \in S$;
- $\sigma_{M^{\mathsf{AML}}} = \{\sigma_M\}$ for all $\sigma \in \Sigma_{s_1\ldots s_n,s}$; note that $\sigma_M\colon M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_s)$ is an element in $[M_{s_1} \to [M_{s_1} \to [\cdots \to [M_{s_n} \to \mathcal{P}(M_s)] \cdots]]]$ under the curry-uncurry isomorphism.

Next, we define the application in M^{AML} as follows:

- $\$ \cdot a = M^{\mathsf{AML}}$ for all $a \in M^{\mathsf{AML}}$;
- $\# \cdot s = M_s$ for all $s \in S$; note that $[\![s_{M^{\mathsf{AML}}}]\!]_{M^{\mathsf{AML}}} = \# \cdot s = M_s$
- $\sigma \cdot a = \{\sigma(a)\}$ for all $\sigma \in [A \to B]$ if B is a function space; and $\sigma \cdot a = \sigma(a)$ for all B of the form $\mathcal{P}(M_s)$, for some $s \in S$.

We split the last case into two cases, depending on if B is a function space. If not, then $\sigma \cdot a = \sigma(a)$ is already a set in $\mathcal{P}(M_s)$. This is different from the corresponding definitions for MSA (Theorem 17) and OSA (Theorem 27).

The unmentioned cases are irrelevant.

We verify that $M^{\mathsf{AML}} \vDash \Gamma^{\mathsf{MSA}}$. (Definedness) and (Nonempty Sort) are satisfied as in Theorem 17. (Arity) for $\sigma \in \Sigma_{s_1\ldots s_n,s}$ is satisfied, because $\sigma_{M^{\mathsf{AML}}}$ is defined as σ_M and application is interpreted as the normal function application.

Finally, we prove that the restricted model of M^{AML} w.r.t. (S, V, Σ), written M^r, is exactly M. By definition, $M^r = (\{M^r_s\}_{s\in S}, \{\sigma_{M^r}\}_{\sigma\in\Sigma})$ is an (S, V, Σ)-model of MmL defined as follows:

- the carrier set $M^r_s = [\![s_{M^{\mathsf{AML}}}]\!]_{M^{\mathsf{AML}}}$, which equals M_s as we showed above, for all $s \in S$;
- the interpretation σ_{M^r}, for $\sigma \in \Sigma_{s_1\ldots s_n,s}$, is defined as $\sigma_{M^r}(a_1, \ldots, a_n) = \sigma_{M^{\mathsf{AML}}} a_1 \cdots a_n$, for all $a_1 \in M_{s_1}, \ldots, a_n \in M_{s_n}$; note that $\sigma_{M^{\mathsf{AML}}} a_1 \cdots a_n = \{\sigma_M\} a_1 \cdots a_n = \sigma_M(a_1, \ldots, a_n)$, so we have $\sigma_{M^r}(a_1, \ldots, a_n) = \sigma_M(a_1, \ldots, a_n)$, for all $a_1 \in M_{s_1}, \ldots, a_n \in M_{s_n}$.

Therefore, M^r is exactly M. □

D.2 Proof of Theorem 22

Theorem 22. *Under the notations of Definition 20 and the above MmL-to-AML translation, any M^r-valuation ρ of MmL derives an M-valuation ρ^{AML} of AML, with $\rho^{\mathsf{AML}}(x^s) = \rho(x{:}s)$ and $\rho^{\mathsf{AML}}(X^s) = \rho(X{:}s)$. Furthermore, $\overline{\rho^{\mathsf{AML}}}(\varphi_s^{\mathsf{AML}}) = \bar{\rho}(\varphi_s)$ for all ρ; and $\Omega^{\mathsf{VALID}} \models \varphi_s^{\mathsf{VALID}}$ iff $\Omega \models_{\mathsf{MmL}} \varphi_s$ for all Ω.*

Proof. For simplicity, we drop s in φ_s and φ_s^{AML} and write φ and φ^{AML}.

We first prove $\overline{\rho^{\mathsf{AML}}}(\varphi^{\mathsf{AML}}) = \bar{\rho}(\varphi)$ by structural induction on φ.

Suppose $\varphi \equiv x{:}s$. Then $\overline{\rho^{\mathsf{AML}}}((x{:}s)^{\mathsf{AML}}) = \overline{\rho^{\mathsf{AML}}}(x^s) = \{\rho^{\mathsf{AML}}(x^s)\} = \{\rho(x{:}s)\} = \bar{\rho}(x{:}s)$.

Suppose $\varphi \equiv X{:}s$. Then $\overline{\rho^{\mathsf{AML}}}((X{:}s)^{\mathsf{AML}}) = \overline{\rho^{\mathsf{AML}}}(X^s) = \rho^{\mathsf{AML}}(X^s) = \rho(X{:}s) = \bar{\rho}(X{:}s)$.

Suppose $\varphi \equiv \sigma(\varphi_1, \ldots, \varphi_n)$. Then $\overline{\rho^{\mathsf{AML}}}((\sigma(\varphi_1, \ldots, \varphi_n))^{\mathsf{AML}}) = \overline{\rho^{\mathsf{AML}}}(\sigma\varphi_1^{\mathsf{AML}} \cdots \varphi_n^{\mathsf{AML}})$
$= \sigma_M \overline{\rho^{\mathsf{AML}}}(\varphi_1^{\mathsf{AML}}) \cdots \overline{\rho^{\mathsf{AML}}}(\varphi_n^{\mathsf{AML}}) = \sigma_M \bar{\rho}(\varphi_1) \cdots \bar{\rho}(\varphi_n) = \sigma_{M^r}(\bar{\rho}(\varphi_1), \ldots, \bar{\rho}(\varphi_n)) = \bar{\rho}(\sigma(\varphi_1, \ldots, \varphi_n))$.

Suppose $\varphi \equiv \varphi_1 \wedge \varphi_2$. Then $\overline{\rho^{\mathsf{AML}}}((\varphi_1 \wedge \varphi_2)^{\mathsf{AML}}) = \overline{\rho^{\mathsf{AML}}}(\varphi_1^{\mathsf{AML}} \wedge \varphi_2^{\mathsf{AML}}) = \overline{\rho^{\mathsf{AML}}}(\varphi_1^{\mathsf{AML}}) \cap \overline{\rho^{\mathsf{AML}}}(\varphi_2^{\mathsf{AML}}) = \bar{\rho}(\varphi_1) \cap \bar{\rho}(\varphi_2) = \bar{\rho}(\varphi_1 \wedge \varphi_2)$.

Suppose $\varphi \equiv \neg\varphi_1$. Then $\overline{\rho^{\mathsf{AML}}}((\neg\varphi_1)^{\mathsf{AML}}) = \overline{\rho^{\mathsf{AML}}}(\neg\varphi_1^{\mathsf{AML}} \wedge [\![s]\!]) = \overline{\rho^{\mathsf{AML}}}((\neg\varphi_1)^{\mathsf{AML}}) \wedge \overline{\rho^{\mathsf{AML}}}([\![s]\!]) = (M \setminus \overline{\rho^{\mathsf{AML}}}(\varphi_1^{\mathsf{AML}})) \cap [\![s_M]\!]_M = [\![s_M]\!]_M \setminus \overline{\rho^{\mathsf{AML}}}(\varphi_1^{\mathsf{AML}}) = M_s^r \setminus \bar{\rho}(\varphi_1) = \bar{\rho}(\neg\varphi_1)$.

Suppose $\varphi \equiv \exists x{:}s\,.\,\varphi_1$. Then $\overline{\rho^{\mathsf{AML}}}((\exists x{:}s\,.\,\varphi_1)^{\mathsf{AML}}) = \overline{\rho^{\mathsf{AML}}}(\exists x^s{:}s\,.\,\varphi_1^{\mathsf{AML}})$
$= \bigcup_{a \in [\![s_M]\!]_M} \overline{\rho^{\mathsf{AML}}[a/x^s]}(\varphi_1^{\mathsf{AML}}) = \bigcup_{a \in M_s^r} \overline{\rho[a/x{:}s]}(\varphi_1) = \bar{\rho}(\exists x{:}s\,.\,\varphi_1)$.

Suppose $\varphi \equiv \mu X{:}s\,.\,\varphi_1$. Then $\overline{\rho^{\mathsf{AML}}}((\mu X{:}s\,.\,\varphi_1)^{\mathsf{AML}}) = \overline{\rho^{\mathsf{AML}}}(\mu X^s\,.\,\varphi_1^{\mathsf{AML}}) = \mu\mathcal{F}^{\rho^{\mathsf{AML}}}_{\varphi_1^{\mathsf{AML}}, X^s}$, where $\mathcal{F}^{\rho^{\mathsf{AML}}}_{\varphi_1^{\mathsf{AML}}, X^s}(A) = \overline{\rho^{\mathsf{AML}}[A/X^s]}(\varphi_1^{\mathsf{AML}})$ for all $A \subseteq M$. Note that for all $A \subseteq M_s^r = [\![s_M]\!]_M \subseteq M$, by inductive hypothesis, $\overline{\rho^{\mathsf{AML}}[A/X^s]}(\varphi_1^{\mathsf{AML}}) = \overline{\rho[A/X{:}s]}(\varphi_1)$. On the other hand, $\bar{\rho}(\mu X{:}s\,.\,\varphi_1) = \mu\mathcal{F}^{\rho}_{\varphi_1, X:s}$ where $\mathcal{F}^{\rho}_{\varphi_1, X:s}(A) = \overline{\rho[A/X{:}s]}(\varphi_1)$ for all $A \subseteq M_s^r$, so $\mathcal{F}^{\rho}_{\varphi_1, X:s}$ and $\mathcal{F}^{\rho^{\mathsf{AML}}}_{\varphi_1^{\mathsf{AML}}, X^s}$ are equal over M_s^r, and thus have the same least fixpoints.

In conclusion, $\overline{\rho^{\mathsf{AML}}}(\varphi_s^{\mathsf{AML}}) = \bar{\rho}(\varphi_s)$ for all M^r-valuations ρ.

Next, we show that $M^r \vDash_{\mathsf{MmL}} \varphi_s$ iff $M \vDash \varphi_s^{\mathsf{VALID}}$. Recall that $\varphi_s^{\mathsf{VALID}} \equiv \psi \rightarrow (\varphi_s^{\mathsf{AML}} = [\![s]\!])$ where $\psi \equiv \bigwedge_{x^{s'} \in \mathrm{FV}(\varphi_s^{\mathsf{AML}})} x^{s'} \in [\![s']\!] \wedge \bigwedge_{X^{s'} \in \mathrm{FV}(\varphi_s^{\mathsf{AML}})} X^{s'} \subseteq [\![s']\!]$.

(Case "if"): Suppose $M \vDash \varphi_s^{\mathsf{VALID}}$ and we want to show that $M^r \vDash_{\mathsf{MmL}} \varphi_s$. Let ρ be any M^r-valuation. Then we know $\rho^{\mathsf{AML}}(\psi) = M$, as ρ^{AML} is derived from ρ and thus it evaluates variables into their corresponding inhabitant sets. Since $M \vDash \varphi_s^{\mathsf{VALID}}$, we have $\overline{\rho^{\mathsf{AML}}}(\varphi_s^{\mathsf{AML}} = [\![s]\!]) = M$, which implies that $\overline{\rho^{\mathsf{AML}}}(\varphi_s^{\mathsf{AML}}) = \overline{\rho^{\mathsf{AML}}}([\![s]\!])$, i.e., $\bar{\rho}(\varphi_s) = M_s^r$. Since ρ is arbitrary, we have $M^r \vDash_{\mathsf{MmL}} \varphi_s$.

(Case "only if"): Suppose $M^r \vDash_{\mathsf{MmL}} \varphi_s$ and we want to show that $M \vDash \varphi_s^{\mathsf{VALID}}$. Let ρ^* be any M-valuation. There are two cases. If there exists a variable $x^{s'}$ (or $X^{s'}$) such that $\rho^*(x^{s'}) \notin M_{s'}^r$ (or $\rho^*(X^{s'}) \not\subseteq M_{s'}^r$), then $\overline{\rho^*}(\psi) = \emptyset$, and thus $\overline{\rho^*}(\varphi_s^{\mathsf{VALID}}) = M$. If otherwise, then all variables evaluate to their corresponding inhabitant sets, and thus there exists an M^r-valuation ρ such that $\rho^* = \rho^{\mathsf{AML}}$. Then, $\overline{\rho^*}(\varphi_s^{\mathsf{AML}} = [\![s]\!]) = M$ iff $\overline{\rho^*}(\varphi_s^{\mathsf{AML}}) = \overline{\rho^*}([\![s]\!])$ iff $\overline{\rho^{\mathsf{AML}}}(\varphi_s^{\mathsf{AML}}) = M_s^r$ iff $\bar{\rho}(\varphi_s) = M_s^r$, which holds by assumption.

In conclusion, $M \vDash \varphi_s^{\mathsf{VALID}}$ iff $M^r \vDash_{\mathsf{MmL}} \varphi_s$.

Finally, we show that $\Omega^{\mathsf{VALID}} \models \varphi_s^{\mathsf{VALID}}$ iff $\Omega \models_{\mathsf{MmL}} \varphi_s$. Recall that $\Omega^{\mathsf{VALID}} = \Gamma^{\mathsf{MmL}} \cup \{\varphi^{\mathsf{VALID}} \mid \varphi \in \Omega\}$.

(Case "if"): Suppose $\Omega \vDash_{\mathsf{MmL}} \varphi_s$ and we want to show that $\Omega^{\mathsf{VALID}} \vDash \varphi_s^{\mathsf{AML}}$. Consider any AML model M such that $M \vDash \Omega^{\mathsf{VALID}}$. Note that $\Omega^{\mathsf{VALID}} \supseteq \Gamma^{\mathsf{MmL}}$. By Theorem 21, its restricted model M^r is an MmL model. We just proved in the above that $M \vDash \varphi_s^{\mathsf{VALID}}$ iff $M^r \vDash_{\mathsf{MmL}} \varphi_s$ for any φ_s. Since $M \vDash \Omega^{\mathsf{VALID}}$, we have $M^r \vDash_{\mathsf{MmL}} \Omega$, which implies $M^r \vDash_{\mathsf{MmL}} \varphi_s$, which implies $M \vDash \varphi_s^{\mathsf{AML}}$. Since M is arbitrary, we have $\Omega^{\mathsf{VALID}} \vDash \varphi_s^{\mathsf{AML}}$.

(Case "only if"). Suppose $\Omega^{\mathsf{VALID}} \vDash \varphi_s^{\mathsf{VALID}}$ and we want to show that $\Omega \vDash_{\mathsf{MmL}} \varphi_s$. Consider any MmL model M^* such that $M^* \vDash_{\mathsf{MmL}} \Omega$. By Theorem 21, there exists an AML model, say M, whose restricted model is exactly M^*; in other words, $M^* = M^r$. Therefore, we have $M^r \vDash_{\mathsf{MmL}} \Omega$, which is equivalent to $M \vDash \Omega^{\mathsf{VALID}}$ as we just proved. By assumption, $M \vDash \varphi_s^{\mathsf{VALID}}$, which then implies that $M^r \vDash_{\mathsf{MmL}} \varphi_s$. Since M^r is an arbitrary MmL model, we have $\Omega \vDash_{\mathsf{MmL}} \varphi_s$. □

E Proof of Proposition 23

Proposition 23. *If $M \vDash \Gamma^{\mathsf{TERM}}$ then the restricted model M^r w.r.t. $(\{Term\}, C)$ is isomorphic to T^C.*

Proof. By Theorem 22 and [11, Proposition 22]. □

F Proof of Proposition 24

Proposition 24. $\Gamma^{\mathsf{NAT}} \vdash zero \in X \wedge (\forall y{:}Nat\,.\, y \in X \to succ\ y \in X) \to \forall x{:}Nat\,.\, x \in X$.

Proof. To prove $\forall x{:}Nat\,.\, x \in X$, we just need to show that $[\![Nat]\!] \subseteq X$. Recall that $[\![Nat]\!] = \mu D\,.\, zero \vee succ\ D$, so we need to show $\mu D\,.\, zero \vee succ\ D \to X$. By (Knaster-Tarski), it suffices to prove $zero \vee succ\ X \to X$, which boils down to proving (1) $zero \to X$ and (2) $succ\ X \to X$. Firstly, (1) is directly proved by $zero \in X$. Secondly, (2) is proved by proving $z \in succ\ X \to z \in X$ for all z, which then boils down to proving $\exists y\,.\, y \in X \wedge z = succ\ y \to z \in X$, which is then proved from $\forall y{:}Nat\,.\, y \in X \to succ\ y \in X$. □

G Proof of Theorem 27

Theorem 27. *$(S, \leq, \Sigma)$-OSA are exactly the restricted Γ^{OSA}-models w.r.t. $(S, \leq, \Sigma)$.*

Proof. All restricted Γ^{OSA}-models w.r.t. $(S, \leq, \Sigma)$ are $(S, \leq, \Sigma)$-OSA, by definition. We just need to show the other direction. For any $(S, \leq, \Sigma)$-OSA, say $A = (\{A_s\}_{s\in S}, \{f_A^{s_1\dots s_n, s}\}_{f^{s_1\dots s_n, s}\in\Sigma})$, we need find an AML model $M \vDash \Gamma^{\mathsf{OSA}}$, whose restricted model w.r.t. $(S, \leq, \Sigma)$ is exactly A.

We first define the carrier set M, by letting it contain/include the following elements/sets:

- \$ for definedness and # for inhabitant;
- S, the sort set;
- A_s, for all $s \in S$;
- $[A_{s_i} \to [A_{s_{i+1}} \to [\cdots \to [A_{s_n} \to A_s]\dots]]]$ for all $\Sigma_{s_1\dots s_n, s} \neq \emptyset$ and $1 \leq i \leq n$;

Next, we define following interpretations of constants:

- $\lceil\_\rceil_M = \{\$\}$ and $[\![\_]\!]_M = \{\#\}$;
- $s_M = \{s\}$ for all $s \in S$;
- $f_M = \{f_A^{s'_1\dots s'_n, s'}\}$ for all $f \in \Sigma_{s_1\dots s_n, s}$, where $f^{s'_1\dots s'_n, s'}$ denotes the overloaded copy of f with the largest arity (w.r.t. subsorting).

Next, we define the application function in M as the following:

- $\$ \bullet a = M$ for all $a \in M$;
- $\# \bullet s = A_s$ for all $s \in S$; note that $[\![s_M]\!]_M = \# \bullet \{s\} = A_s$;
- $f \bullet a = \{f(a)\}$ for all $f \in [A \to B]$ and $a \in A$.

The unmentioned cases are irrelevant.

Now we verify that M satisfies all axioms in Γ^{OSA}. Indeed, we only need to consider (Subsort), which is satisfied, because $[\![s]\!]_M = A_s \subseteq A_{s'} = [\![s']\!]_M$, for all $s \leq s'$. Therefore, $M \vDash \Gamma^{\mathsf{OSA}}$.

Finally, we prove that the restricted model M^r w.r.t. $(S, \leq, \Sigma)$ is exactly A. By definition, $M^r = (\{M^r\}_{s\in S}, \{f_{M^r}\}_{s\in S})$ is an $(S, \leq, \Sigma)$-OSA defined as follows:

- the carrier set $M^r_s = [\![s_M]\!]_M$, which equals A_s as we showed above, for all $s \in S$;

- the interpretation $f_{M^r}^{s_1\dots s_n,s}$, for every $f^{s_1\dots s_n,s} \in \Sigma_{s_1\dots s_n,s}$, is defined such that $\{f_{M^r}^{s_1\dots s_n,s}(a_1,\dots,a_n)\} = f_M a_1 \cdots a_n$ for all $a_1 \in A_1,\dots,a_n \in A_n$; note that $f_M a_1 \cdots a_n = \{f_A^{s_1'\dots s_n',s'}\} a_1 \cdots a_n = \{f_A^{s_1'\dots s_n',s'}(a_1,\dots,a_n)\} = \{f_A^{s_1\dots s_n,s}(a_1,\dots,a_n)\}$, because $f^{s_1\dots s_n,s}$ and $f^{s_1'\dots s_n',s'}$ are subsort overloaded. Therefore, we have $f_{M^r}(a_1,\dots,a_n) = f_A(a_1,\dots,a_n)$ for all $a_1 \in A_1,\dots,a_n \in A_n$.

Therefore, M^r is exactly A. □

H Proof of Proposition 32

Proposition 32. *For all models M satisfying the axioms in Definition 31, $[\![s^2]\!]_M \cong [\![s]\!]_M^2$.*

Proof. Let $\langle \_, \_ \rangle_M$ denote the interpretation of pairing in M, and we write $\langle \_, \_ \rangle_M \cdot a \cdot b$ as $\langle a, b \rangle_M$ for $a, b \in [\![s]\!]_M$. Note that $\langle \_, \_ \rangle$ is defined as a function, so it always returns singletons on arguments in $[\![s]\!]_M$ and by abuse of notation we use $\langle a, b \rangle_M$ to also denote the element in the singleton set. By (Product Set) axiom, $[\![s^2]\!]_M = \bigcup_{a \in S, b \in S} \langle a, b \rangle_M$, so there is a natural surjective function $i \colon [\![s]\!]_M^2 \to [\![s^2]\!]_M$ defined as $i(a, b) = \langle a, b \rangle_M$. To prove $[\![s^2]\!]_M = [\![s]\!]_M^2$, we just need to show that i is injective, which follows directly from (Injectivity). □

I Proof of Proposition 34

Proposition 34. *For all models M satisfying the axioms in Definition 33, $[\![2^s]\!]_M \cong \mathcal{P}([\![s]\!]_M)$.*

Proof. Let ext_M denote the interpretation of extension in M, and we write $ext_M \cdot A$ as $ext_M(A)$ for all $A \in [\![2^s]\!]_M$. By the powerset semantics of AML, we know $ext_M(\_) \colon [\![2^s]\!]_M \to \mathcal{P}([\![s]\!]_M)$ is a function, so we just need to prove that it is injective and surjective. The injectivity follows from (Extensionality), because for $A, B \in [\![2^s]\!]_M$ with $A \neq B$, we have $ext_M(A) \neq ext_M(B)$. The surjectivity follows from (Powerset), because for any $C \in \mathcal{P}([\![s]\!]_M)$, given as the valuation of the free set variable X in (Powerset), there exists $A \in [\![2^s]\!]_M$ such that $ext_M(A) = C$. □

By abuse of language, we also write $int_M(A)$ to mean the "interpretation" of the syntactic sugar *int*. Under the isomorphism given in Proposition 34, $ext_M(A) = int_M(A) = A$, for all $A \subseteq [\![s]\!]_M$. Note that in $ext(A)$, A is considered an element while the result, also equals to A, is considered as a set. Such a distinction between elements and sets are important in evaluating AML patterns, because of its powerset semantics and, in particular, the *pointwise extension* over sets. In other words, when A is regarded as a set, the pointwise extension happens; otherwise, A is simply an element, and no pointwise extension is needed.

J Proof of Theorem 35

We tacitly assume the isomorphism between a function $f \colon A \to B$ and its graph $\mathsf{graph}(f) = \{(a, f(a)) \mid a \in A\}$. Note that a graph is a subset of $A \times B$, but not vice versa: not all subsets of $A \times B$ are graphs. In this sense, the sort $Graph \equiv 2^{Pair}$ defined in Section 8.4, contains all binary relation over *Term*, including those that are graphs and those are not. However, we are only interested in those that are graphs.

Theorem 35. $\vdash_\lambda e_1 = e_2$ *iff* $\Gamma^\lambda \vdash e_1 = e_2$.

Proof. The "only if" direction is by (1) equational reasoning holds in AML (Proposition 13); and (2) Γ^λ contains all instances of (β). Here we just need to prove the "if" direction.

We follow the approach in Fig. 4, where the only nontrivial step is Step 3, as shown below:

$$M \vDash e_1 = e_2 \text{ for all AML models } M \vDash \Gamma^\lambda \text{ implies } M \vDash_\lambda e_1 = e_2 \text{ for all concrete ccc models } M.$$

Indeed, Step 1 holds by the soundness of AML (Theorem 14); Steps 2 and 4 are definitions; and Step 5 holds by the completeness of concrete ccc models (Lemma 30).

Our proof consists of two parts. Firstly, we show that for any concrete ccc model M, we can define an AML model, written M^{AML}, such that $M^{\mathsf{AML}} \vDash \Gamma^\lambda$. Secondly, we show that for any valuation ρ of λ-calculus, we can define

an M^{AML}-valuation, written ρ^{AML}, such that $\overline{\rho^{\mathsf{AML}}}(e) = \{|e|_\rho\}$ for all $e \in \Lambda$. Given these two results, the rest of the proof is simple. Suppose the assumption holds. Consider an arbitrary concrete ccc model M. We want to prove that $M \vDash_\lambda e_1 = e_2$. For that, we consider an arbitrary valuation ρ and try to prove $|e_1|_\rho = |e_2|_\rho$. By the above two results, it suffices to prove $\overline{\rho^{\mathsf{AML}}}(e_1) = \overline{\rho^{\mathsf{AML}}}(e_2)$, which holds by the assumption.

Let us assume a concrete ccc model $M = \{M, \_\cdot\_, \mathbb{G}\}$, with $R(M) \subseteq [M \to M]$ contains all representable functions and $\mathbb{G}\colon R(M) \to M$ is the retraction function, and we define M^{AML} as follows. Firstly, we define its carrier set by letting contain/include all the following elements/sets:

- \$ for definedness and # for inhabitant;
- M, for sort *Term*;
- $M \times M$, for sort *Pair*;
- $\mathcal{P}(M \times M)$, for sort *Graph*;
- all proper function spaces for interpreting $\langle \_, \_ \rangle$, *ext*, and G as (partial) functions, as well as their partial applications, as in Theorem 17;

Next, we define the interpretation of constants and application in M^{AML} such that:

- $\lceil \_ \rceil_{M^{\mathsf{AML}}} = \{\$\}$ and $\$ \cdot a = a$ for all $a \in M^{\mathsf{AML}}$;
- $[\![\_]\!]_{M^{\mathsf{AML}}} = \{\#\}$, with $\# \cdot \mathit{Term} = M$, $\# \cdot \mathit{Pair} = M \times M$, and $\# \cdot \mathit{Graph} = \mathcal{P}(M \times M)$;
- $\langle \_, \_ \rangle_{M^{\mathsf{AML}}}$ is defined as the standard pairing function, $\mathit{ext}_{M^{\mathsf{AML}}}$ is defined as the identity function over $\mathcal{P}(M \times M)$, and $G_{M^{\mathsf{AML}}}$ is defined as $\mathbb{G}$;
- $f \cdot a = f(a)$, whenever f is a function and a is an argument in its domain.

In particular, the following hold:

- $\mathit{ext}_{M^{\mathsf{AML}}} \cdot A = A$ for $A \in \mathcal{P}(M \times M)$; note that the left A is considered an element of $\mathcal{P}(M \times M)$, so pointwise extension does not apply;
- $G_{M^{\mathsf{AML}}} \cdot A = \mathbb{G}(f_A)$ for $f \in R(M)$ and $A = \mathsf{graph}(f_A)$; similarly, A is considered as an element, so no pointwise extension happens;
- For notational simplicity, we write $\mathit{int}_{M^{\mathsf{AML}}}(A)$ to mean $\bar{\rho}(\mathit{int}\ X)$ for an M^{AML}-valuation ρ with $\rho(X) = A$. By definition, we can prove that $\mathit{int}_{M^{\mathsf{AML}}}(A) = A$, for $A \subseteq M \times M$.

We verify that $M^{\mathsf{AML}} \vDash \Gamma^\lambda$. Indeed, all axioms except (β) are satisfied, following the same reasoning as in Theorem 17, Proposition 32, and Proposition 34. We postpone proving $M^{\mathsf{AML}} \vDash (\beta)$ and prove the following result first. After that, $M^{\mathsf{AML}} \vDash (\beta)$ follows easily.

In the following, we prove that $\overline{\rho^{\mathsf{AML}}}(e) = \{|e|_\rho\}$ for all $e \in \Lambda$, where ρ^{AML} is an M^{AML}-valuation, defined as $\rho^{\mathsf{AML}}(x) = \rho(x)$ for all variables x of λ-calculus, which are also element variables of M^{AML}. The values of ρ^{AML} on other variables are not irrelevant. We carry out structural induction on e as follows:

- Suppose $e \equiv x$. Then $\overline{\rho^{\mathsf{AML}}}(x) = \{\rho^{\mathsf{AML}}(x)\} = \{\rho(x)\}$;
- Suppose $e \equiv e_1 e_2$. Then $\overline{\rho^{\mathsf{AML}}}(e_1 e_2) = \overline{\rho^{\mathsf{AML}}}(e_1)\overline{\rho^{\mathsf{AML}}}(e_2) = \{|e_1|_\rho\}\{|e_2|_\rho\} = \{|e_1|_\rho |e_2|_\rho\} = \{|e_1 e_2|_\rho\}$.
- Suppose $e \equiv \lambda x.e$. Then $\overline{\rho^{\mathsf{AML}}}(\lambda x.e) = \overline{\rho^{\mathsf{AML}}}(G\ (\mathit{int}\ \exists x\colon \mathit{Term}.\langle x, e\rangle))$
  $= G_{M^{\mathsf{AML}}} \cdot \overline{\rho^{\mathsf{AML}}}(\mathit{int}\ \exists x\colon \mathit{Term}.\langle x, e\rangle) = G_{M^{\mathsf{AML}}} \cdot \mathit{int}_{M^{\mathsf{AML}}}(\overline{\rho^{\mathsf{AML}}}(\exists x\colon \mathit{Term}.\langle x, e\rangle)) = G_{M^{\mathsf{AML}}} \cdot \overline{\rho^{\mathsf{AML}}}(\exists x\colon \mathit{Term}.\langle x, e\rangle) =$
  $G_{M^{\mathsf{AML}}} \cdot \bigcup_{a \in M} \overline{\rho^{\mathsf{AML}}[a/x]}(\langle x, e\rangle) = G_{M^{\mathsf{AML}}} \cdot \bigcup_{a \in M}(\{a\}, \overline{\rho^{\mathsf{AML}}[a/x]}(e))$
  $= G_{M^{\mathsf{AML}}} \cdot \bigcup_{a \in M}(\{a\}, \{|e|_{\rho[a/x]}\}) = G_{M^{\mathsf{AML}}} \cdot \mathsf{graph}(f^\rho_{e,x}) = \mathbb{G}(f^\rho_{e,x}) = |\lambda x.e|_\rho$.

Therefore, $\overline{\rho^{\mathsf{AML}}}(e) = \{|e|_\rho\}$ for all $e \in \Lambda$. And here ends our proof. □

$$\text{(START)}\ \frac{\Gamma \vdash_{\mathsf{TS}} A\ \mathit{type}}{\Gamma, x{:}A \vdash_{\mathsf{TS}} x{:}A} \qquad \text{(WEAK)}\ \frac{\Gamma \vdash_{\mathsf{TS}} A\ \mathit{type} \quad \Gamma, \Gamma' \vdash_{\mathsf{TS}} \Theta}{\Gamma, x{:}A, \Gamma' \vdash_{\mathsf{TS}} \Theta}$$

$$\text{(PRIM)}\ \Gamma \vdash_{\mathsf{TS}} \tau\ \mathit{type} \quad \text{if } \tau \in \mathsf{PType}$$

$$\text{(}\Pi\text{)}\ \frac{\Gamma, x{:}A \vdash_{\mathsf{TS}} B\ \mathit{type}}{\Gamma \vdash_{\mathsf{TS}} \Pi x{:}A.B\ \mathit{type}}$$

$$\text{(}\Pi\text{-I)}\ \frac{\Gamma, x{:}A \vdash_{\mathsf{TS}} b{:}B \quad \Gamma, x{:}A \vdash_{\mathsf{TS}} B\ \mathit{type}}{\Gamma \vdash_{\mathsf{TS}} \lambda x{:}A.b : \Pi x{:}A.B} \qquad \text{(}\Pi\text{-E)}\ \frac{\Gamma \vdash_{\mathsf{TS}} a{:}\Pi x{:}A.B \quad \Gamma \vdash_{\mathsf{TS}} b{:}A \quad \Gamma, x{:}A \vdash_{\mathsf{TS}} B\ \mathit{type}}{\Gamma \vdash_{\mathsf{TS}} ab{:}B[b/x]}$$

$$\text{(}\Sigma\text{)}\ \frac{\Gamma, x{:}A \vdash_{\mathsf{TS}} B\ \mathit{type}}{\Gamma \vdash_{\mathsf{TS}} \Sigma x{:}A.B\ \mathit{type}} \qquad \text{(}\Sigma\text{-I)}\ \frac{\Gamma \vdash_{\mathsf{TS}} a{:}A \quad \Gamma \vdash_{\mathsf{TS}} b{:}B[a/x] \quad \Gamma, x{:}A \vdash_{\mathsf{TS}} B\ \mathit{type}}{\Gamma \vdash_{\mathsf{TS}} \mathit{pair}\ a\ b : \Sigma x{:}A.B}$$

$$\text{(}\Sigma\text{-E1)}\ \frac{\Gamma \vdash_{\mathsf{TS}} a : \Sigma x{:}A.B \quad \Gamma \vdash_{\mathsf{TS}} A\ \mathit{type}}{\Gamma \vdash_{\mathsf{TS}} \mathit{fst}\ a : A} \qquad \text{(}\Sigma\text{-E2)}\ \frac{\Gamma \vdash_{\mathsf{TS}} a : \Sigma x{:}A.B \quad \Gamma, x{:}A \vdash_{\mathsf{TS}} B\ \mathit{type}}{\Gamma \vdash_{\mathsf{TS}} \mathit{snd}\ a : B[(\mathit{fst}\ a)/x]}$$

Figure 7: Typing rules of the Martin-Löf type system, as give in [58]; (here Θ is $b{:}B$ or $B\ \mathit{type}$)

K Proof of Theorem 37

Theorem 37. *If* $x_1{:}A_1, \ldots, x_n{:}A_n \vdash_{\mathsf{PTS}} a{:}A$ *then* $\Gamma^{\mathsf{PTS}} \vdash x_1{:}A_1 \wedge \cdots \wedge x_n{:}A_n \to a{:}A$.

Proof. Indeed, the four axioms in Γ^{PTS} capture the PTS typing rules (Π), (Π-I), (Π-E), and (AXIOM), respectively. The rest rules (START), (WEAK), (CONV) can be generically proved in AML, where the first two are by standard FOL reasoning and the last is by equational reasoning plus (β). □

L Instance: Martin-Löf type system

Here we show how to define the *Martin-Löf type system* [36] as an AML theory. The Martin-Löf type system is an extension of λ-calculus with *types*, built from primitive types and type constructors. It has many variants; here we consider the one in [58], focusing on two important type constructors: the Π-*types* $\Pi x{:}A.B$ and the Σ-types $\Sigma x{:}A.B$.

Given a set VAR of *variables* $x, y, \ldots$ and a set PType of *primitive types* $\tau_1, \tau_2, \ldots$, the Martin-Löf type system defines *types* and *terms* as follows

$$\begin{aligned} &\text{types}\ \ A, B \coloneqq \tau \in \mathsf{PType} \mid \Pi x{:}A.B \mid \Sigma x{:}A.B \\ &\text{terms}\ \ a, b \coloneqq x \in \text{VAR} \mid \lambda x{:}A.b \mid ab \mid \mathit{pair}\ a\ b \mid \mathit{fst}\ a \mid \mathit{snd}\ a \end{aligned}$$

where Π, Σ, and λ are binders that bind x into B or b, but not A. Compared to PTS, $\Sigma x{:}A.B$ is a new type constructor that builds Σ-types, whose elements are pairs, built from *pair*, and can be destructed with *fst* and *snd*. As in PTS (Section 9), a *typing context* is a sequence $x_1{:}A_1, \ldots, x_n{:}A_n$ for $n \geq 0$. We use ϵ to denote the empty context. The (β) axiom is also assumed. The typing rules of the Martin-Löf type system are shown in Fig. 7, most of which are similar to PTS typing rules in Fig. 5. They derive two kinds of *judgments*: $\Gamma \vdash_{\mathsf{TS}} a{:}A$, which specifies that a has type A under Γ, and $\Gamma \vdash_{\mathsf{TS}} A\ \mathit{type}$, which specifies that A is a type under Γ. For notational simplicity, when we write down $\Gamma \vdash_{\mathsf{TS}} a{:}A$ and $\Gamma \vdash_{\mathsf{TS}} A\ \mathit{type}$, we mean the claim that they can be proved in the Martin-Löf type system.

Now we define an AML theory Γ^{TS} that captures the Martin-Löf type system. Firstly, we define two sorts: *Term* for terms and *Type* for types. For every primitive type $\tau \in \mathsf{PType}$, we define a corresponding functional constant and define the axiom

$$\text{(PRIMITIVE TYPE)}\quad \tau\ \mathit{Type} \qquad /\!/ \text{ where we define } A\ \mathit{type} \equiv A \in [\![\mathit{Type}]\!]$$

Next, we define λ, Π, and Σ as binders, using the generic method discussed in Section 8.5. Next, we define three new constants, *pair*, *fst*, and *snd*, each defining its counterpart in the Martin-Löf type system. Finally, we let Γ^{TS} contain

(β), the typed version, (Primitive Type) for all $\tau \in \mathsf{PType}$, plus the following self-explanatory typing axioms:

(Π) $\forall x{:}A\,.\,B\ \mathit{type} \to \Pi x{:}A\,.\,B\ \mathit{type}$ (Π-I) $\forall x{:}A\,.\,b{:}B \wedge \forall x{:}A\,.\,B\ \mathit{type} \to \lambda x{:}A\,.\,b{:}\Pi x{:}A\,.\,B$

(Π-E) $a{:}\Pi x{:}A\,.\,B \wedge b{:}A \wedge \forall x{:}A\,.\,B\ \mathit{type} \to ab{:}B[b/x]$

(Σ) $\forall x{:}A\,.\,B\ \mathit{type} \to \Sigma x{:}A\,.\,B\ \mathit{type}$ (Σ-I) $a{:}A \wedge b{:}B[a/x] \wedge \forall x{:}A\,.\,B\ \mathit{type} \to \mathit{pair}\ a\ b{:}\Sigma x{:}A\,.\,B$

(Σ-E1) $a{:}\Sigma x{:}A\,.\,B \wedge A\ \mathit{type} \to \mathit{fst}\ a{:}A$ (Σ-E2) $a{:}\Sigma x{:}A\,.\,B \wedge \forall x{:}A\,.\,B\ \mathit{type} \to \mathit{snd}\ a{:}B[(\mathit{fst}\ a)/x]$

These axioms capture the corresponding typing rules in Fig. 7. As in PTS, some rules are automatically captured by the generic AML reasoning, so no axioms for them are needed in Γ^{TS}.

Theorem 42. *Let* $\Gamma \equiv x_1{:}A_1, \ldots, x_n{:}A_n$ *and* $\psi_\Gamma \equiv x_1{:}A_1 \wedge \cdots \wedge x_n{:}A_n$. *Then we have* $\Gamma \vdash_{\mathsf{TS}} \Theta$ *implies* $\Gamma^{\mathsf{TS}} \vdash \psi_\Gamma \to \Theta$, *for all* Θ *of the form* $a{:}A$ *or* A *type.*

Proof. Indeed, the three rules for Π-types and four rules for Σ-types in Fig. 7 are captured by the corresponding axioms in Γ^{TS}, respectively. (Prim) is captured by the (Primitive Type) axiom. The rest two rules, (Start) and (Weak), are captured by generic AML reasoning. □

M Proof of Proposition 39

Proposition 39. *Given a theory* Γ *and an* injective *and* extensional *context* c, *i.e.,* $\Gamma \vdash c[x] \wedge c[y] = c[x \wedge y]$, $\Gamma \vdash c[\bot] = \bot$ *and* $\Gamma \vdash c[x \vee y] = c[x] \vee c[y]$, *then* $\Gamma \vdash c[x \Rightarrow y]$ *implies* $\Gamma \vdash c[x] \Rightarrow c[y]$.

Proof. The proof is basic AML reasoning, Recall that $\varphi_1 \Rightarrow \varphi_2 \equiv \varphi_1 \to \bullet\varphi_2$.

Firstly, we have $\Gamma \vdash c[x \Rightarrow y]$ implies $\Gamma \vdash c[x \to \bullet y]$, which implies $\Gamma \vdash c[\neg x \vee \bullet y]$, which implies $\Gamma \vdash c[\neg x] \vee c[\bullet y]$ by extensionality, which implies $\Gamma \vdash \neg c[\neg x] \to c[\bullet y]$. On the other hand, we need to prove $\Gamma \vdash c[x] \Rightarrow c[y]$, i.e., $\Gamma \vdash c[x] \to \bullet c[y]$, which is implied by $\Gamma \vdash c[x] \to c[\bullet y]$ by (Lift Rewrites). Therefore, it suffices to prove $\Gamma \vdash c[x] \to \neg c[\neg x]$, which is implied by $\Gamma \vdash \neg(c[x] \wedge c[\neg x])$ by FOL reasoning, which is implied by $\Gamma \vdash \neg(c[x \wedge \neg x])$ by injectivity, which is implied by $\Gamma \vdash \neg c[\bot]$, which is implied by $\Gamma \vdash \neg\bot$ by extensionality, which is implied by $\Gamma \vdash \top$, which holds. □

N Proof Checker Code

We show the full code of the AML proof checker in the next two pages. Fig. 8 shows the infrastructure code, including those defining AML pattern syntax, free variables, substitution, α-equivalence, as well as proof rules and proof objects. Fig. 9 shows the proof checking code consisting of 22 equations, which follow blindly the proof system of AML.

```
module CHECKER is protecting NAT .

sorts EVar SVar Var VType VSet . subsorts EVar SVar < Var .
ops ev sv : -> VType . mb ev(N) : EVar . mb sv(N) : SVar .
op _`(_`) : VType Nat -> Var .
sort Sigma . op cs : Nat -> Sigma .
sort Pat . subsorts Var Sigma < Pat .
ops \app \imp : Pat Pat -> Pat . op \bot : -> Pat . ops \ex \mu : Nat Pat -> Pat .

subsort Var < VSet . op . : -> VSet . op __ : VSet VSet -> VSet [assoc comm id: . ] .
op _\_ : VSet VSet -> VSet . op _in_ : Var VSet -> Bool .
var V : Var . vars VT VT' : VType . vars Vs Vs' : VSet . vars N N' N'' M : Nat . vars P Q R P' Q' R' : Pat .
eq V V = V .
eq (V Vs) \ (V Vs') = Vs \ Vs' . eq Vs \ Vs' = Vs [owise] .
eq V in V Vs = true . eq V in Vs = false [owise] .

op fresh : VSet -> Nat .
eq fresh(.) = 0 . eq fresh((VT(N)) Vs) = 1 + max(N, fresh(Vs)) .

op fv : Pat -> VSet .
eq fv(VT(N)) = VT(N) . eq fv(cs(N)) = . . eq fv(\bot) = . .
eq fv(\app(P,Q)) = fv(P) fv(Q) .
eq fv(\imp(P,Q)) = fv(P) fv(Q) .
eq fv(\ex(N,P)) = fv(P) \ ev(N) .
eq fv(\mu(N,P)) = fv(P) \ sv(N) .

op _[_/_] : Pat Pat Var -> Pat .
eq VT(N)[R / VT(N)] = R . eq VT(N)[R / VT'(N')] = R [owise] .
eq \app(P,Q)[R / VT(N)] = \app(P[R / VT(N)], Q[R / VT(N)]) .
eq \imp(P,Q)[R / VT(N)] = \imp(P[R / VT(N)], Q[R / VT(N)]) .
eq \ex(N, P)[R / ev(N)] = \ex(N, P) . eq \mu(N, P)[R / sv(N)] = \mu(N, P) . eq \bot[R / VT(N)] = \bot .
ceq \ex(N', P)[R / VT(N)] = \ex(N'', P[ev(N'') / ev(N')][R / VT(N)]) if N'' := fresh(fv(P) fv(R)) .
ceq \mu(N', P)[R / VT(N)] = \mu(N'', P[sv(N'') / sv(N')][R / VT(N)]) if N'' := fresh(fv(P) fv(R)) .

op _=a=_ : Pat Pat -> Bool .
eq VT(N) =a= VT(N') = N == N' . eq \bot =a= \bot = true . eq P =a= Q = false [owise] .
eq \app(P,Q) =a= \app(P',Q') = P =a= P' and Q =a= Q' . eq \imp(P,Q) =a= \imp(P',Q') = P =a= P' and Q =a= Q' .
ceq \ex(N,P) =a= \ex(N',P') = P[ev(N'') / ev(N)] =a= P'[ev(N'') / ev(N')] if N'' := fresh(fv(P) fv(P')) .
ceq \mu(N,P) =a= \mu(N',P') = P[sv(N'') / sv(N)] =a= P'[sv(N'') / sv(N')] if N'' := fresh(fv(P) fv(P')) .

sorts Thm Rule . op `(_`)_by_ : Nat Pat Rule -> Thm [prec 90] .
op axiom : -> Rule .
ops p1 p2 p3 : -> Rule .
op mp`(_,_`) : Nat Nat -> Rule .
op ex`(_`) : Nat -> Rule .
op ug : -> Rule .
ops ppbotL ppbotR pporL pporR ppexL ppexR : -> Rule .
ops frmL`(_`) frmR`(_`) : Nat -> Rule .
op existence : -> Rule .
op singleton-ev`(_,_`) : Pos Pos -> Rule .
op sv-subst`(_,_`) : Nat Pat -> Rule .
op prefixpoint : -> Rule .
op kt`(_`) : Nat -> Rule .

sort Pos . ops . l r : -> Pos . op __ : Pos Pos -> Pos [assoc id: .] . op _[_] : Pat Pos -> Pat .
vars Pos PosP PosQ : Pos .
eq P[ . ] = P . eq \app(P,Q)[ l Pos ] = P[Pos] . eq \app(P,Q)[ r Pos ] = Q[Pos] .

sorts Thms . subsort Thm < Thms . op . : -> Thms . op __ : Thms Thms -> Thms [assoc id: . prec 100] .
vars RL RL' RL'' : Rule . vars PTs QTs RTs PTs' QTs' RTs' PTs'' : Thms .

sort Proof .
op proof_ : Thms -> Proof [prec 110] . op check : -> Thm . op checked : -> Proof .
```

Figure 8: Proof checker (Part 1): infrastructure code

```
eq proof PTs check = checked .

eq proof PTs check (M) P by axiom QTs = proof PTs (M) P by axiom check QTs .

eq proof PTs check (M) \imp(P, \imp(Q, P)) by p1 QTs = proof PTs (M) \imp(P, \imp(Q, P)) by p1 check QTs .

eq proof PTs check (M) \imp(\imp(P, \imp(Q, R)), \imp(\imp(P, Q), \imp(P, R))) by p2 QTs
 = proof PTs (M) \imp(\imp(P, \imp(Q, R)), \imp(\imp(P, Q), \imp(P, R))) by p2 check QTs .

eq proof PTs check (M) \imp(\imp(\imp(P, \bot), \bot), P) by p3 QTs
 = proof PTs (M) \imp(\imp(\imp(P, \bot), \bot), P) by p3 check QTs .

eq proof PTs (N) P by RL PTs' (N') \imp(P, Q) by RL' PTs'' check (M) Q by mp(N,N') QTs
 = proof PTs (N) P by RL PTs' (N') \imp(P, Q) by RL' PTs'' (M) Q by mp(N,N') check QTs .

eq proof PTs (N') \imp(P, Q) by RL' PTs' (N) P by RL PTs'' check (M) Q by mp(N,N') QTs
 = proof PTs (N') \imp(P, Q) by RL' PTs' (N) P by RL PTs'' (M) Q by mp(N,N') check QTs .

ceq proof PTs check (M) \imp(R, \ex(N, P)) by ex(N') QTs
  = proof PTs (M) \imp(R, \ex(N, P)) by ex(N') check QTs if R =a= P[ev(N') / ev(N)] .

ceq proof PTs (N) \imp(P, Q) by RL PTs' check (M) \imp(\ex(N, P), Q) by ug QTs
  = proof PTs (N) \imp(P, Q) by RL PTs' (M) \imp(\ex(N, P), Q) by ug check QTs if not(ev(N) in fv(Q)) .

eq proof PTs check (M) \imp(\app(\bot, P), \bot)  by ppbotL QTs
 = proof PTs (M) \imp(\app(\bot, P), \bot)  by ppbotL check QTs .

eq proof PTs check (M) \imp(\app(P, \bot), \bot)  by ppbotR QTs
 = proof PTs (M) \imp(\app(P, \bot), \bot)  by ppbotR check QTs .

eq proof PTs check (M) \imp(\app(\imp(\imp(P, \bot), Q), R), \imp(\imp(\app(P, R), \bot), \app(Q, R))) by pporL QTs
 = proof PTs (M) \imp(\app(\imp(\imp(P, \bot), Q), R), \imp(\imp(\app(P, R), \bot), \app(Q, R))) by pporL check QTs .

eq proof PTs check (M) \imp(\app(R, \imp(\imp(P, \bot), Q)), \imp(\imp(\app(R, P), \bot), \app(R, Q))) by pporR QTs
 = proof PTs (M) \imp(\app(R, \imp(\imp(P, \bot), Q)), \imp(\imp(\app(R, P), \bot), \app(R, Q))) by pporR check QTs .

ceq proof PTs check (M) \imp(\app(\ex(N, P), Q), \ex(N, \app(P, Q))) by ppexL QTs
  = proof PTs (M) \imp(\app(\ex(N, P), Q), \ex(N, \app(P, Q))) by ppexL check QTs if not(ev(N) in fv(Q)) .

ceq proof PTs check (M) \imp(\app(P, \ex(N, Q)), \ex(N, \app(P, Q))) by ppexR QTs
  = proof PTs (M) \imp(\app(P, \ex(N, Q)), \ex(N, \app(P, Q))) by ppexR check QTs if not(ev(N) in fv(P)) .

eq proof PTs (N) \imp(P, P') by RL PTs' check (M) \imp(\app(P,Q),\app(P',Q)) by frmL(N) QTs
 = proof PTs (N) \imp(P, P') by RL PTs' (M) \imp(\app(P, Q), \app(P', Q)) by frmL(N) check QTs .

eq proof PTs (N) \imp(Q, Q') by RL PTs' check (M) \imp(\app(P, Q), \app(P, Q'))  by frmR(N) QTs
 = proof PTs (N) \imp(Q, Q') by RL PTs' (M) \imp(\app(P, Q), \app(P, Q'))  by frmR(N) check QTs .

eq proof PTs check (M) \ex(N, ev(N)) by existence QTs = proof PTs (M) \ex(N, ev(N)) by existence check QTs .

ceq proof PTs check (M) \imp(P, \imp(Q, \bot)) by singleton-ev(PosP, PosQ) QTs
  = proof PTs (M) \imp(P, \imp(Q, \bot)) by singleton-ev(PosP, PosQ) check QTs
 if \imp(\imp(V:EVar,\imp(R,\bot)),\bot) := P[PosP] /\ \imp(\imp(V':EVar, R'), \bot) := Q[PosQ]
 /\ V:EVar == V':EVar /\ R == R' .

ceq proof PTs (N) Q by RL PTs' check (M) P by sv-subst(N, R) QTs
  = proof PTs (N) Q by RL PTs' (M) P by sv-subst(N, R) check QTs if P =a= Q[R / sv(N)] .

ceq proof PTs check (M) \imp(Q, \mu(N, P)) by prefixpoint QTs
  = proof PTs (M) \imp(Q, \mu(N, P)) by prefixpoint check QTs if Q =a= P[\mu(N, P) / sv(N)] .

ceq proof PTs (N) \imp(Q, R) by RL PTs' check (M) \imp(\mu(N, P), R) by kt(N) QTs
  = proof PTs (N) \imp(Q, R) by RL PTs' (M) \imp(\mu(N, P), R) by kt(N) check QTs if Q =a= P[R / sv(N)] .

endmodule
```

Figure 9: Proof checker (Part 2): proof checking code